4

Technical Document 1924
October 1990

Evolutionary Software Engineering at the Naval Postgraduate School (NPS), Monterey

Luqi

Approved for public release; distribution is unlimited.

The views and conclusions contained in this report are those of the contractors and should not be interpreted as representing the official policies, either expressed or implied, of the Naval Ocean Systems Center or the U.S. Government.

NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152-5000

J. D. FONTANA, CAPT, USN
Commander

R. M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

This work was performed for the Naval Ocean Systems Center, San Diego, CA 92152-5000, under program element 0602234N. The work was carried out by Luqi, Assistant Professor of Computer Sciences, Naval Postgraduate School, Computer Science Department, Monterey, CA 93943, under the technical coordination of W. L. Sutton, Computer Systems and Software Technology Branch, Code 411, Naval Ocean Systems Center.

Released by
D. L. Hayward, Head
Computer Systems and
Software Technology Branch

Under authority of
A. G. Justice, Head
Information Processing
and Displaying Division

MODELS FOR EVOLUTIONARY SOFTWARE DEVELOPMENT

Luqi

Models for Evolutionary Software Development

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

This report explores the benefits to be derived from elaborating and automating models for evolutionary software development. The structure of the report is summarized in Fig. 1.

1. Introduction

In recent years, the software engineering research community has been focusing significant attention on the process of software development and enhancement as a complement to the more traditional emphasis on the products of those processes. This interest recently has led to several new approaches to modeling and analyzing software processes [1-6] which provide ways of coping with some of the difficulties encountered in previous models.

In view of the rapidly expanding number of competing approaches, it is important to re-assess the key issues of software development processes, re-examine some fundamental assumptions, identify common characteristics of life-cycle models, and determine

* The Problem
* Objectives
* Key Issues
* Advantages
* Approach
* Example

Fig. 1 Outline

some principles guiding the steps of software processes.

Examples of models for evolutionary software development are provided in the attached additional technical reports, as summarized below.

(1) NPS 52-88-39 "Software Evolution Via Prototyping", describes rapid prototyping models used in evolutionary software development.

(2) NPS 52-89-16 "Petri-Net Based Models of Software Engineering Process" presents an extension of the classical Petri net model to formally define the functional, structural, and dynamic aspects of software engineering processes.

(3) NPS 52-89-44 "Software Analysis and Testing Through Prototyping" focuses on the validation and verification aspect of the software development process, and indicates the role of prototyping methods in this context.

(4) NPS 52-89-56 "Multi-level Software Analysis and Testing in Evolutionary Software Development" discusses research directions on aspects of software analysis and testing in the software process, and identifies needed software analysis processes at all levels of the development process, from requirements analysis to software evolution.

1.1. Problems With Models

Some of the difficulties encountered in previous models are shown in Fig. 2. The variety of incomparable models makes it difficult to compare and evaluate different software development processes, and to combine different models which focus on different aspects of software development. Alternative paradigms for development make comparisons difficult because concepts important in one model may not have any counterparts in another model based on a completely different paradigm. Accurate comparisons are impossible ·vhen the basic concepts and notations defining a model have not been clearly defined, and experimental evaluation becomes impossible when actual development practices do not correspond to the models used to describe and analyze those

*** the variety of incomparable models,**

*** alternative underlying software development paradigms,**

*** lack of precise meaning of concepts and notation used, and**

*** mismatch between models and practical development practices.**

Fig. 2 Difficulties With Previous Models

processes.

Some of these issues are illustrated in Fig. 3, which shows a plausible but informal description of the software development process. Some questions raised by this example are shown in Fig. 4. An essential problem with previous models has been that they are too informal. Early models were described by simple block diagrams without any underlying mathematical structure or precise definitions. Such models may be useful for orientation purposes, but they do not provide enough structure to form the basis for computer-aided software development.

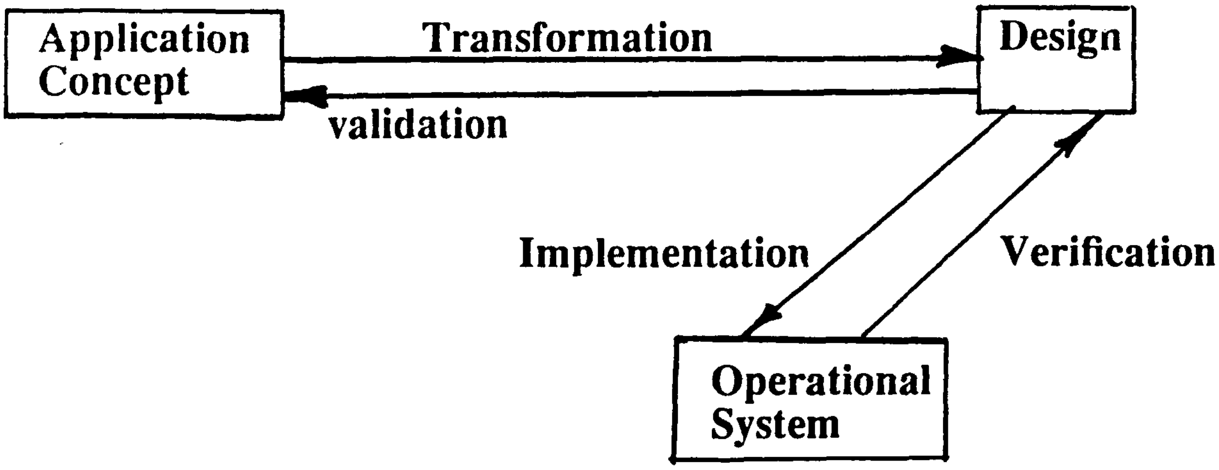


Fig. 3 Example of an Imperfect Process Model

* What does the notation mean (box, arrow, text)?

* How are these elements composed?

* How can complete observation of such a model be guaranteed?

Fig. 4 Questions Raised by the Imperfect Process Model

Practical software development processes must cope with the difficulties shown in Fig. 5.

Most software products are developed for a group of users with differing needs and concerns. These concerns may conflict with each other, and are imprecisely known because most of the potential users of the system do not interact directly with the developer. In addition to being uncertain, the goals for a large system can be very complex, to the point where a single person may not be able to understand them all in detail. Currently software objects, processes, and tools are not completely understood, partially because of unresolved technical and scientific questions, and partially because all three are constantly being changed and extended. In the face of so much uncertain information, feedback is essential for avoiding wasted effort and reducing risk. Such feedback is needed early in the process to provide the opportunity to solve potential problems before

*** uncertain, complex requirements**

*** a variety of user concerns**

*** incomplete knowledge about software objects, software processes, and tools**

*** insufficient use of validated, prefabricated, & adaptable software components**

*** risks of misdevelopment due to late or insufficient feedback information**

*** individuality of application domains, organizations, methods and tools implies need to adapt processes**

*** long lifetime of software requires enhancements due to changing requirements and environmental conditions**

*** need to integrate development and maintenance processes performed by different organizations**

Fig. 5 Difficulties of Software Development

the allotted time for the project has run out. Software development processes must be adaptable to different applications and methods because the conditions for different development projects can vary widely. This implies that software development is not a single process, but rather a family of related processes with a rich structure and many conceptual dependencies. Finally, the long lifetime of software products and the large fraction of the costs associated with software evolution is a major concern. Practical software development must accommodate many modifications or enhancements to the product as the process proceeds, and it must address the issue of handing the product over to a maintenance organization once the initial development is complete.

2. The Problem

The need to model and analyze software engineering processes is more important today than ever, because the advent of new methods and technologies aimed at various aspects of these processes is forcing managers and developers to decide how to best utilize them. The variety of life-cycle and process models presents a problem. Different approaches are hard to compare and judge for the reasons listed in Fig. 6.

Most of the life-cycle models do not have sufficient empirical data to prove their effectiveness and show their impact on software quality. In the mass of competing approaches much of the common sense and many of the basic principles of software engineering processes hidden in technical details of specific process models need to be

*** emphasize different aspects of software development processes and thereby are likely to sacrifice others,**

*** use different concepts and notations, and**

*** support different software development paradigms such as**

- automatic programming and formal program transformation,

- evolutionary development via rapid prototyping and fourth generation languages, and

- 'knowledge-based software assistant' approaches.

Fig. 6 Difficulties in Comparing Different Approaches

re-assessed. Theoretical foundations and analytical comparisons are needed in this area because it is very expensive to develop a life cycle model, create the necessary tool support and carry out experiments to determine the relative effectiveness of competing approaches in practice. The conceptual foundations of the field must be clarified to the point where meaningful questions can be asked, and appropriate experiments can be designed to provide useful answers to those questions with a relatively small number of case studies.

3. Objectives

The objectives for future work on software process models should include the ones listed in Fig. 7.

A meta-model is needed to provide a formal framework and precise notations for formalizing and comparing alternative approaches. Such formalization is needed to support meaningful comparisons and automated tools for effectively realizing the process models in practice. Prototyping is a promising new approach for supporting evolutionary software development, which provides a useful and challenging test case for exercising the meta-model. A meat-model should be capable of describing development methods and supporting tools with sufficient detail to enable automated monitoring or control functions which are capable of preventing or detecting errors and recording derivations or justifications for design decisions based on the process model. The model should provide the basis for automating the aspects of development concerning coordination of component activities and interactions between the different people or processes involved. The process model should also aid in the analysis and description of the product, and provide guidance for the design and construction of automated engineering database support for analyzing and describing the resulting software products.

4. Key Issues

Some of the key issues in the proposed approach are identifying and separating the goals of the meta-model and the goals of the software development process models it will define. The meta-model should provide the capabilities shown in Fig. 8.

A comprehensive and standardized vocabulary is needed to allow meaningful comparisons of different process models. In addition to standardized terminology, standardized structures for describing the elements of software development processes are needed. Such elements include engineering data, states of a development project, actions transforming states and data, constraints on the order of the actions, and the mechanisms for carrying out the actions or verifying that they have been correctly carried out.

Software process models will be defined using the facilities provided by the meta-model. Software process models derived should have the properties listed in Fig. 9.

A process model can be subjected to useful scientific analysis and automated procedures for supporting the model can be objectively designed only if the process model has a precise formal representation. A flexible representation scheme is needed, because the process often has to be changed as it is carried out in response to new information and new circumstances. Management considerations indicate that the model should provide measurable properties of the process which can support estimation, planning, and quality control activities in an objective and computer-aided way. The process models should be

* Defining a meta-model of software processes which

- supports alternative development paradigms,

- reflects common understanding and fundamental characteristics of software processes,

- accommodates enhancements to an ongoing process,

- facilitates effective management of development processes,

- enables automation

* Designing a concrete process model supporting evolutionary software development through prototyping.

* Describing suitable methods and support tools.

* Preventing/detecting errors.

* Recording derivation/justification

* Providing foundations for automating part of the process, reducing coordination problems and increasing speed.

* Analyzing & documenting the product model with the assistance of design/project/engineering databases.

Fig. 7 Objectives for Research

* A comprehensive vocabulary.

* A small set of structuring principles
to describe the elements and structure of

- data domains,

- development states,

- actions transforming software objects from one state into another

- mechanisms that help affect such transformations
on different levels of abstraction and from different perspectives.

Fig. 8 Properties of a Meta-Model

A software process model should

* be formal and explicit to enable automated consistency and completeness analysis, reasoning, and replay

* be executable to support symbolic testing and a priori demonstration of designed development processes

* be changeable as they are executed, to respond to new information

* be measurable for estimation, planning, quality control

* be parameterized to reflect dependencies on properties of the application

* organize tools to guide coordination of development tasks and define units of configuration control

* provide a coherent framework for communication, management, and tool development

* provide reusable standard process components with alternatives

Fig. 9 Properties of a Software Process Model

parameterized to capture the dependencies of the process model on the properties of the application domain, so that the necessary adaptation can be carried out in a disciplined and predictable way. The process model provides the context for organizing tools and coordinating development tasks in terms of meaningful transactions which can serve as the basis for configuration control. By making the process predictable and providing a systematic structure for representing alternative choices, formalized process models provide a basis for communication, project management, tool development, and the construction of reusable libraries of standardized process components. This should provide better control over the development process and should enable effective tailoring of the process to current needs of particular projects.

5. Advantages

The advantages of the proposed approach are summarized in Fig. 10.

It is very difficult to make accurate cost and schedule estimates if the tasks to be carried out are unknown. The process model provides a structure for predicting and identifying the tasks involved in particular projects, thus aiding estimation. A clear picture of

*** Schedules and costs of development steps can be better calculated based on precise knowledge of actions involved.**

*** Resources can be better allocated based on knowledge about causal dependencies among actions.**

*** Development history of individual components can be traced.**

*** Alternatives courses of development are made explicit.**

*** Process model execution can be automated.**

*** Alternative development steps can be evaluated prior to process execution.**

*** Different process models can be compared and put into contrast.**

*** Fundamental characteristics of software processes are reflected by the basic building blocks and structuring mechanisms of the meta-model.**

*** Families of process models are constructed via parameterization and (de-)composition mechanisms.**

*** Process histories are formally deduced from process models.**

Fig. 10 Advantages of the Proposed Approach

causal dependencies between actions can make planning more systematic and can enable automated procedures for aiding decision makers in evaluating their options. The development history of a product can be presented in meaningful terms if it can be linked to a coherent process model. The development history of a software product contains a huge amount of information, which is useless unless it can be structured and simplified to make it comprehensible and to enable people to find the particular pieces of information about the past which will help them make meaningful decisions about what to do in the next step. The formal structure can enable automated support for identifying and evaluating the alternative choices a designer has at each point in the development. The structure also allows evaluation of different process models, and can provide decision support tools for determining which process model is most appropriate for a particular project. The structures provided by generic process components with parameters, explicit composition mechanisms, and standard sets of building blocks enable concise descriptions of different alternatives, which make it easier to represent alternatives and to identify choices. Similar structures induced on process histories can be used for re-evaluating decisions when the product must be modified, by locating and organizing the relevant design choices in terms of the structure of the development processes involved. These structures can also be useful in diagnosing and locating errors.

6. Approach

Our approach to meeting these objectives consists of the tasks shown in Fig. 11.

*** survey of life-cycle models and software development methods used for evolutionary software development,**

*** design of a meta-model by exploiting experience from software specification and design techniques,**

*** specification and restructuring of a selected prototyping approach to software development in terms of the meta-model,**

*** adaptation and integration of prototyping tools to support the designed process, and**

*** evaluate analysis and reasoning capabilities.**

Fig. 11 Summary of Tasks

A survey of existing process models provides an initial version of the requirements for a meta-model, by providing a set of test cases. The proposed meta-model should be capable of representing all of the concerns addressed by current informal process models. These concerns must be formalized and their essential features abstracted to provide a clean and independent set of building blocks for the meta-model. The proposed meta-model should be exercised, evaluated, and extended by applying it to the formalization and improvement of a selected prototyping approach to evolutionary software development. This will involve restructuring and integrating the prototyping tools to correspond to the developed process model. The application will provide a basis for evaluating the analysis and reasoning capabilities provided by the tools.

7. Example

A simplified example of a generalized meta-model is shown in Fig. 12. According to this model, the software development process consumes resources and produces a software product. Different versions of the model differ in the types of resources that are considered and the detailed composition of the resulting products, as well as the steps that are carried out to create the products.

7.1. Versions

The generalized model has multiple specializations, all of which fit the same general framework. Each of the specializations represents an alternative approach to software development. Reasons for using multiple alternatives are listed in Fig. 13.

7.2. Prototyping

The prototyping cycle is one possible variation of the meta-model. Prototyping is an attractive approach for situations described in Fig. 14. We focus on this variation in our case study because it covers an important class of applications.

7.3. Evaluation

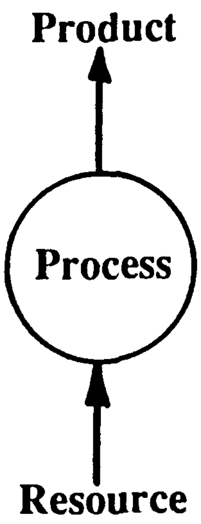
Some of the important characteristics of prototyping are shown in Fig. 15.

Prototyping seeks to reduce costs and errors by providing inexpensive feedback earlier in the development process. This is accomplished by separating concerns and reusing software components at different levels of the process. Separation of concerns enhances the capabilities for incremental analysis, development, and validation.

7.4. Functions of Prototyping

The roles of prototyping in software development are shown in Fig. 16.

It is difficult to communicate with users about a proposed new system because software is abstract and difficult to visualize. Since most users are not specialists in computer science, they cannot be expected to understand formal notations for describing system behavior. Demonstrations of the behavior of a prototype provide a representation of the proposed system's behavior that users can readily understand and evaluate. Since a prototype is constructed quickly and inexpensively, it can provide user feedback early in the development process, when adjustments and modifications have a much lower cost than near the end of the cycle. A prototype can provide a demonstration of the feasibility of implementing key system concepts, and can provide the basis for evaluating the merits



Product = { Version }

Version = requirements + spec + design
+ code + manuals + ...

Resource = budget + time + people
+ hardware + tools
+ software components + ...

Fig. 12 Representative/Meta/Generalized Software Development Model

Different

* types of applications,

* tool sets, and

* development organizations.

Fig. 13 Reasons for Adapting Process Models

* unfamiliar applications,

* systems that change user organizations, and

* complex/hard real-time/embedded systems.

Fig. 14 Applicability of Prototyping

* reduced costs and errors,
* quick small feedback loops,
* separation of concerns,
* different levels of reuse, and
* incremental development.

Fig. 15 Characteristics of Prototyping

* communication with users,
* quick & inexpensive feedback,
* demonstration of feasibility,
* evaluation of alternative designs,
* aid in synthesis: decomposition, and
* platform for evolution: flexibility.

Fig. 16 Roles of Prototyping

of alternative designs for critical subsystems. A prototype can help in the construction of the production-quality system by helping to arrive at a modular decomposition of the problem. The prototype also provides a basis for evolution, because prototypes are typically described in simple, high level notations and provide a simplified view of the system before optimization transformations have introduces additional details and logical dependencies that reduce flexibility.

7.5. Creating a CAPS

Prototyping is most effective if supported by a computer-aided prototyping system (CAPS). Such a system is a mechanism for speeding up the process. Some of the important characteristics of a CAPS are shown in Fig. 17.

A formal prototyping language is needed to serve as a basis for the automated tools comprising the CAPS. Such a language should provide simplicity and expressive power to make it easy to analyze and process prototype descriptions and allow a designer to rapidly construct a prototype with minimal mental effort. Such a language is used as a medium to formulate proposed system behaviors in a form that can be demonstrated and measured by automated tools. The language is used to specify and document the intended behavior of the system, both for guiding later development steps and as a basis

* a formal prototyping language,

* basis for automated tools,

* provides simplicity & expressive power,

* formulate, demonstrate, measure,

* specify and document,

* link to reusable components,

* computer-aided transformation to implementation,

* execution Support: simulate, translate, schedule, monitor,

* user interface: graphics, syntax-directed edit, browser,

* database: configuration + reusable components, and

* optimization: refinement + transformation.

Fig. 17 Capabilities of a Prototyping System

for automatically retrieving reusable software components that can help to realize the prototype. The specification part of the prototyping language should also serve as the basis for computer-aided transformations into production-quality implementations.

The tools for execution support should provide capabilities to simulate components that are not yet available and to translate descriptions of system decompositions into reasonably efficient implementations. While performance of a prototype is not an overriding issue, prototypes must run with sufficient speed to provide demonstrations of system behaviors within practical time periods. Scheduling is needed to evaluate the feasibility of meeting real-time constraints in a proposed system, relative to given estimates of design characteristics for the system. Facilities for monitoring the execution of the prototype are critical for evaluation and debugging purposes. User interface considerations are important for speeding up the process of constructing and modifying a prototype. Graphical displays, syntax directed editing facilities which provide support for computer-assisted design completion, and browsers for quickly locating relevant pieces of information are some of the kinds of tools that can provide support for user interface issues. Database support is also critical for effective rapid prototyping, because of the large volumes of information involved. Some of the critical functions of the database portion of a CAPS are configuration control and management of reusable components. Tools for refinement and optimization are essential for helping to cope with performance issues. Such issues arise when transforming a prototype design into a production-quality implementation, and when the performance of the prototype must be improved to enable effective demonstrations.

7.6. Levels of Analysis and Testing

A prototype serves as a vehicle for analyzing and testing proposed systems at the earliest stages of requirements analysis and functional specification. Most approaches to testing require the generation of system outputs or responses by some means. Three approaches to providing this capability are summarized in Fig. 18, along with indications of the advantages and disadvantages of each approach. Simulation involves some form of direct execution of a component specification. The advantage of this approach is low designer effort, because the specification is a simplified view of the component, which leaves out many details. This approach has the disadvantage of inefficiency because the details that are left out are needed for efficient execution. General methods for evaluating specifications, such as equation solving or logic programming, usually involve inherently slow processes such as exhaustive, unbounded searching.

An interpreter for a module interconnection language provides better efficiency than direct execution of specifications, and provides generally good flexibility and control because the execution support system has access to detailed information about the execution of the prototype and its relationships to the formal prototype description. Some disadvantages of this approach are that it requires some additional designer effort and that the timing of the prototype execution does not reflect the timing characteristics of the production version of the system very accurately. This second disadvantage is shared with the simulation approach.

The approach of translating the prototype description directly into Ada has the advantages of providing relatively accurate timing estimates, better efficiency, and access to many reusable software components. Disadvantages of this approach are difficulties in

Simulation: Executable spec

+ low designer effort

- inefficient

Interpreter: Module interconnection language

+ Flexible: dynamic binding & modification

+ Controllable: powerful debug, reverse execution

- Timing does not reflect production version

Ada: Augment and transform specifications

+ Efficient

+ Accurate timing

+ Reusable components

- Hard to construct

- Inflexible

Fig. 18 Evaluation of Approaches to Prototype Execution

modifying the properties of the prototype as it executes, and difficulties in constructing the prototype, because of the extra details the designer must specify.

Since none of these approaches is clearly superior to the others, a CAPS should support all three possibilities, and allow them to be used together in the same prototype.

7.7. Verification and Validation

Verification and validation are essential parts of the prototyping process, as illustrated in Fig. 19. The aspect of a prototype design that is most important to verify is the correctness of a proposed system decomposition.

Testing means simulating the design using a finite set of test cases to see if the proposed structure operates as inteded. Testing is an effective means of detecting errors, but it is usually not capable of certifying correctness of a design. In cases where reliability is critical, mechanically checked proofs of correctness may be needed. Such proofs involve showing that a given interconnection of specified components realize the specified behavior of the subsystem for all possible inputs. Such proofs are simpler than traditional proofs of correctness because they operate entirely at the specification and design level, without any consideration of coding details.

The other major function of prototyping is to validate proposed system behavior. Demonstrations allow the user to inspect actual system behaviors and determine whether they meet the real needs of the user, rather than some approximation to those needs captured by a written specification. To effectively carry out such validations, it is necessary to identify a set of typical operational transactions or scenarios which illustrate actual problems that are supposed to be solved by the proposed system. Such transactions act as test cases to be used in the user demonstration. These test cases must be mapped into the interactions supported by the prototype. In cases where this mapping cannot be carried out, faults in the system are detected without the need for users to be involved in a demonstration.

Verification

* testing, or

* proof at component spec level.

Validation

* user demonstrations, and

* typical operational transactions.

Fig. 19 Verification and Validation via Prototyping

7.8. Configuration Control

The need for speed in developing prototypes implies that more than one designer will usually be involved in the process. The purpose of the configuration management facilities of a CAPS is to help coordinate group activities as detailed in Fig. 20. It is essential that modifications made by one designer do not get lost or invalidate work done by other designers. Configuration control mechanisms meet these goals by enforcing serializability of updates via some type of locking protocol. Configuration control systems are subject to additional goals of minimizing the amount of lost time due to waiting for locks.

Another function of configuration control is to avoid wasting everyone's time by making a damaged version of a subsystem visible to the entire group. This goal can be partially met by automatically running mechanical error checking procedures before making a version public, and requiring repairs before allowing a version into the baseline in case the mechanical checks fail.

Configuration control for prototypes provides software objects with frozen versions. The motivation for this capability is shown in Fig. 21.

*** Non-interference: locking, serializability, and**

*** Correctness: enforce error checking.**

Fig. 20 Configuration Control in Prototyping

*** stability,**

*** no read locks,**

*** creating new alternatives: no write locks, and**

*** computer-aided merging of alternatives.**

Fig. 21 Advantages of Frozen Versions

Since versions cannot change, a prototype constructed from a given set of versions of its subsystems provides a stable and reproducible snapshot of a design alternative. Such a snapshot can be recreated and evaluated as necessary, without concern for interference by ongoing experimentation with alternative versions of the prototype. Since versions cannot change, there is no need for locks preventing the reading of a version, thus producing maximum access to all existing versions. If the configuration management system supports creating new alternative lines of development, then there is no need for write locks either, and all designers are free to create new versions without the need to worry about interference. This is made possible by creating a new alternative line of development whenever a lock would have blocked access in an ordinary database system. Such a facility must be combined with automated support for recombining or merging the features of alternative lines of development. This process must be reliable, in the sense that all potential conflicts need to be detected by the merging process.

7.9. Derivations: Evolution and Variations

The database for supporting rapid prototyping should provide facilities for capturing and utilizing derivation information for the decisions embodied in a prototype design. Desirable properties of such derivations are shown in Fig. 22.

The logical structure of a design history indicates what decisions were made and the logical dependencies between them. This information is different than the historical order in which decisions were made, because independent decisions should not be artificially ordered by historical accident. Also, logical dependencies between decisions should not be hidden just because they were made out of order by mistake. The logical structure of a design history should indicate the refinements in each line of development, where each refinement corresponds to the information added by a compatible design decision. Alternatives represent incompatible ways of resolving the same aspect of a design, and represent choice points for the designer. Separating logical dependencies from historical orderings and explicitly representing alternative choices enables computer support for reordering decisions and factoring out common parts of different lines of development. This allows the system to simplify and clarify the choices faces by the

*** logical structure, not actual history,**

*** refinements and alternatives, and**

*** computer-aided reordering and factoring.**

Fig. 22 Properties of Derivation Histories

designer, and makes it easier to navigate through the design space when seeking to modify an evolving system.

7.10. Versions of Composite Objects

Prototypes of practical software systems are too large to be built as monolithic structures, and hence are usually realized by some type of hierarchical decomposition. This introduces the problem of managing the versions of composite objects, which is described in Fig. 23.

A change to the specification of a composite subsystem introduces a natural unit of atomic transaction: the change should be made either completely or not at all. This induces some dependencies between the versions of the components of the modified subsystem. A configuration control system should support such atomic transactions by keeping track of which versions of the components at the next lower level correspond to each version of the composite subsystem. This facility is important for exploring design alternatives quickly, because it allows the designer to switch between alternative versions of a high level subsystem without concern for the dependencies between the subcomponents: these are managed automatically by the configuration control system.

8. Conclusion

More than 25 theses at NPS show the feasibility of different aspects of computer aided prototyping. Improved solutions for many problems are desirable. Many of these solutions involve interactions between different aspects of the software development process. We have found the need for better formulations of the structure of the development process to provide more effective tools support and to guide the further development of computer-aided prototyping systems.

1. B. Boehm, "A Spiral Model of Software Development and Enhancement", *Computer 21*, 5 (May 1988), 61-72.
2. B. Curtis, H. Krasner and N. Iscoe, "A field study of the software design process for large systems", *Comm. of the ACM 31*, 11 (Nov. 1988), 1268-1287.
3. A. Finkelstein, " "Not waving but drowning": Representation Schemes for modelling software development", in *Proceedings of the 11th Annual*

*** atomic transactions**

*** coordinating versions of components**

Fig. 23 Managing Versions of Composite Objects

International Conference on Software Engineering, Pittsburgh. PA, May 1989, 402-404.

4. W. Humphrey and M. Kellner, ''Software process modelling: Principles of entity process models'', in *Proceedings of the 11th Annual International Conference on Software Engineering*, Pittsburgh, PA, May 1989, 331-342.

5. B. Kraemer, *Concepts, Syntax and Semantics of SEGRAS - A Specification Language for Distributed Systems*, Oldenbourg Verlag,, Muenchen-Wien, 1989.

6. Luqi, ''Software Evolution via Rapid Prototyping'', *IEEE Computer*, May 1989.

DISTRIBUTION LIST

(1) Defense Technical Information Center 2
Cameron Station
Alexandria, Virginia 22304-6145

(2) Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Monterey, CA 93943

(3) Center for Naval Analysis 1
4401 Ford Avenue
Alexandria, VA 22302-0268

(4) Director of Research Administration 1
Attn: Prof. Howard
Code 012
Naval Postgraduate School
Monterey, CA 93943

(5) Chairman, Code 52 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

(6) Chief of Naval Research 1
800 N. Quincy Street
Arlington, Virginia 22217

(7) National Science Foundation 1
Division of Computer and Computation Research
Attn. Tom Keenan
Washington, D.C. 20550

(8) Naval Postgraduate School 50
Code 52Lq
Computer Science Department
Monterey, CA 93943

(9) Attn: Linwood Sutton 1
Naval Ocean Systems Center
Code 411
San Diego, CA 92152-5000

MULTI-LEVEL SOFTWARE ANALYSIS AND TESTING IN EVOLUTIONARY SOFTWARE DEVELOPMENT

Luqi

Multi-Level Software Analysis and Testing in Evolutionary Software Development

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

This paper presents a view of research directions in multi-level software analysis and testing. Software analysis is needed at different levels of the development process. Appropriate research goals are identified for exploring software analysis at each of these levels in order to produce good quality software at reduced cost.

1. Introduction

The goals of software analysis and testing are to measure essential software properties and to enable development of software systems with specified ranges for those properties. Reliability of software products is gaining increasing importance, particularly for systems whose malfunction may result in loss of human life, compromise of national security, or massive loss of property. Software properties related to reliability include constraints on functional behavior, timing, and storage space. Software maintenance is also a major concern because it typically accounts for more than half the cost of a software system. Software properties related to maintenance include the subset of a software system affected by a proposed change, the effects of a software modification on a given reliability property, and invariant relationships among members of a family of software systems.

Classical approaches to reliability are based on the assumption of an imperfect software development process, and hence focus on detecting errors and measuring reliability properties of software products. More recently proposed approaches explore special types of computer-aided software development processes which can guarantee the resulting software products are free of particular classes of errors via properties of the formalized development process. Research on software analysis and testing should address processes for achieving desired properties of software products as well as methods for measuring those properties.

2. Levels of Software Analysis and Testing

Different types of software analysis and testing are appropriate at different stages of software development, as summarized in Fig. 1.

2.1. Requirements Level

The requirements level established the goals for a proposed system and formulates models of the problem and the expected environment of the proposed system.

Level	Type of Analysis and Testing
Requirements	consistency: truth maintenance
	model validation: simulation and proof
	subgoal verification: prototyping and proof
Specification	adequacy: prototyping, operational scenarios
	consistency: type and domain checking
	safety: proofs
	validation: paraphrasing, views, simplification
Design	verification: proof of decomposition
	liveness: deadlock and starvation checking
	robustness: impact of degraded hardware
	design for testing: control and observation
	performance: complexity analysis
	feasibility: satisfiability proofs
Coding	synthesis: meaning-preserving transformations
	performance: time and space analysis
	liveness: proof of (clean) termination
	real-time: analysis of scheduling methods
	generic units: analysis of component families
	error detection: complete test sets
	error location: weakest preconditions
Evolution	change impact: symbolic differences
	restructuring: meaning-preserving transformations

Fig. 1 Types of Software Analysis and Testing

An important aspect of requirements analysis is achieving and maintaining consistency as the analysts discover and record the requirements. A promising approach to this problem is providing automated support for calculating and maintaining derived properties and consequences of the requirements, and for tracing dependencies to determine the causes of conflicts and inconsistencies. Better algorithms for this process and primitives suitable for expressing and effectively maintaining dependencies in software requirements should be investigated.

Another aspect of requirements analysis is modeling the environment of a proposed system. Especially for embedded software systems, an accurate formal characterization of the system to be controlled is essential for assessing the effectiveness of the control software. The environment of such a system must often be simulated or otherwise fonnally analyzed to enable safe and meaningful testing or analysis of the embedded software system. Systematic methods for validating and testing the formal models of the environment against the properties of the actual physical systems they represent are needed. Both analytical and experimental methods should be explored to establish that the formal environment models used in other software analysis and testing activities are adequate representations of reality.

Many critical software systems are embedded systems, which means that the software is part of a larger system. Thus an essential part of checking the adequacy of the requirements for the software is checking that any system meeting the requirements will be sufficient to meet the requirements for the larger system in its intended operational context. Hard real-time constraints in an embedded system are often motivated by the requirement to control the larger system to ensure it remains within a given range of operating states. For example, the cycle rate of an auto-pilot must be sufficiently high to ensure that the airplane remains within a given radius of its planned position at all times. At the current time, lower level requirements are usually formulated based on past experience and informal guidelines rather than on systematic derivations or verification procedures with respect to the higher level requirements. Both formal and experimental methods for systematically establishing such properties are needed. Required supporting technology for this process includes computer-aided construction of prototypes.

2.2. Specification Level

The specification level is concerned with defining the interface of a proposed system, both at the functional and the command representation levels.

The primary measure of the adequacy of a specified interface is whether it will meet the needs of the user. This question is best addressed by experimental rather than analytical techniques because it addresses the problem of checking the correspondence between a formalized specification and the actual and informal needs of the users. One way of approaching this problem is via prototyping and operational scenarios. Operational scenarios are common tasks in the customer's problem domain, expressed in the user's terms. Such scenarios serve as test cases for the specifications, whose purpose is to determine whether a proposed interface is adequate for carrying out all of the tasks the users will have to perform. Such a test passes if the facilities provided by the proposed system interface can be combined to carry out the tasks in the operational scenario, and provide a systematic means for exercising a prototype in a demonstration to the users. Systematic methods for deriving sets of scenarios from a requirements document, coverage criteria, and experimental evaluation of the effects of such coverage criteria on change requests to the affected interfaces during system maintenance should be investigated.

A related concern is validating a formal specification, to ensure that it correctly captures the intentions of the users. While this is an informal process, it can be aided by formalized and automatable procedures. Some of the processes involved are paraphrasing, projection, and simplification. Paraphrasing is the process of transforming a formal specification into a form that a user can understand, while preserving its meaning. Projection is the process of extracting the parts of a specification relevant to a particular user or task, while hiding other details. Simplification is the process of transforming a formal specification into a simpler form with an equivalent meaning. These three processes can be combined to help users selectively review formal specifications using representations they can understand. The research questions in this area concern certifying the transformations to ensure they preserve the meaning of the specification and experimentally evaluating the effectiveness of different representations for communicating with untrained users.

Consistency of a specification is another common concern, especially for large and complex systems. Since consistency is a property of a formal document, it can be addressed by analytical techniques. Some aspects of consistency checking that need further development are type and domain consistency checking. Type checking at the specification level is more difficult than the corresponding problem at the code level because types can have subtypes defined by semantic considerations. Domain checking is the process of ensuring that partial functions or predicates are used only within their domain of definition, and that partially defined generic units are instantiated only with actual parameters in their respective domains of definition. Logical inference capabilities are necessary for both of these kinds of specification analysis.

Another concern with formal specifications is checking safety properties. For example, past research proiects have been concerned with whether a proposed operating systems kernel satisfied certain security properties, such as the impossibility of transmitting classified information from a process with a high security classification to an unauthorized process. The goals of safety analysis procedures are to identify cases where the specifications allow behaviors violating the safety properties, or to certify that no such cases exist. Systematic procedures for this process are needed because the connection between a formal specification and a safety property can be quite indirect and can require extensive reasoning and analysis to establish.

2.3. Design Level

The design level is concerned with the decomposition of a problem into a hierarchical structure of independent modules. Such a decomposition consists of interconnection information and formal specifications for the components.

The primary reliability property of a decomposition structure is whether it will correctly realize the specification at the next higher level. This problem is subject to mathematical proof techniques. The problem is easier to solve than the general proof of correctness problem at the code level because the module interconnection language is can be considerably simpler than a programming language. Most of the analysis can be carried out at the specification level, since the problem is to check whether a given combination of specified components will satisfy the required properties of the composite. Research questions in this area involve the best choice of interconnection primitives to support effective and efficient inference procedures.

Another type of property of interest for parallel and distributed systems is liveness. Techniques for checking for potential deadlock or starvation conditions in such a design are desired. Such techniques can be based on the combination of fast graph algorithms with satisfiability checking for paths leading to potential problems. The main research questions are finding efficient special purpose analysis techniques that can address semantic issues which are neglected by classical Petri net techniques or involve infinite graphs if encoded as standard Petri nets.

An important class of analysis involves the effects of degraded hardware on the properties of a design, relative to a mapping of software components to hardware components. This kind of analysis is essential for achieving reliable fault tolerant systems, especially those with distributed implementations. In addition to certifying that proposed configurations realize given degrees of fault tolerance, automatic derivation of the implied constraints on allocation of software functions to hardware units is desirable.

Automated assessment or augmentation of a design for supporting testing is another issue at the design level. Facilities for control and observation of closed modules such as abstract data types and machines are needed to support testing. Guidelines for what attributes of a system need to be controlled or monitored for effective testing must be developed, along with automated techniques for generating the code that realizes the control and monitoring functions to be added during testing. Such an investigation should be coupled with an analysis of the impact of the additional code on time and space requirements, and techniques for automatically compensating for their effects in checking timing and space constraints.

Evaluation of a design for time and space performance is another kind of software analysis that has potential importance. Automated support for classical complexity analysis is needed, along with estimates for the ranges of input sizes and constant factors determined by classes of algorithms.

A final consideration is satisfiability. The satisfiability of a specification can be established if an implementation can be produced and certified to be correct. However, it would be useful to determine whether it is possible to satisfy a given specification before the implementation is attempted, and in cases where it is not, to characterize the set of inputs for which the specification is impossible to meet. Analytical techniques for constructing weakest infeasible preconditions characterizing *this set of inputs* should be explored.

2.4. Code Level

The best way to achieve quality is to systematically prevent errors. Automatable methods for synthesis of efficient code from formal specifications via meaning-preserving transformations should be investigated. Of particular interest are systems that can choose transformations without explicit human guidance, or with guidance from general declarative advice that can be formulated without explicit reference to the details of the current state of the derivation and does not require explicit human interaction during the derivation process.

Accurate performance analysis requires detailed code and knowledge about properties of a particular compiler and target machine. Generic table driven methods for performing such analysis, and for relating design-level properties of abstract algorithms to detailed properties of actual machine-level implementations and compiler optimizations is needed to accurately certify correctness of programs with hard real-time and real-space constraints. Research problems in this area include formal modeling of implementation-specific properties and constraints in ways that can be combined with implementation-independent analyses of abstract programs.

Another problem is certification of clean termination. This problem gains new dimensions in parallel and distributed systems, where termination can be influenced by scheduling properties and hardware failures. Research questions include models and techniques for analyzing programs in these domains.

Analysis of real-time systems includes analysis of scheduling methods to determine whether a proposed scheduling discipline will meet specified deadlines under all possible operating conditions. Research questions in this area have flexible scheduling methods, the effects of shared resources, overload resolution policies, and remote communications as major concems.

The problem of certifying generic code units or families of related programs generated by meta-programming schemes is a major concem in systems for managing reusable software. A software component is most effectively re-used if it is flexible and can be adapted to many needs. Such a component often corresponds to a family of related program units with an unbounded number of elements. The problems of testing and analyzing the reliability of such program families is an important research question.

Classical testing approaches need more foundational work on the construction of complete test sets. A complete test set is a set of test cases which is guaranteed to detect any error in a particular well-defined class of errors. More work is needed on the construction of finite complete test sets, an on characterizing the set of faults whose absence is guaranteed by successful execution of the test set. Such work should include automated techniques for constructing the required test oracles from the formal specifications of the code to be tested.

An aspect of code analysis of great practical importance is error location. One approach to this problem is to derive weakest preconditions for suspected pieces of code, to characterize the space of inputs for which the code fails.

2.5. Evolution Aspects

Software maintenance is acknowledged to be more difficult and error prone than the initial development. An important kind of software analysis for this part of software development is characterization of the effects of a change to a software system. Symbolic representations for the parts of the input space and the output space of a program affected by a given change to the code are useful for testing and evaluating a modification for conformance with the expected results. Computer-aided identification of the parts of a specification affected by a given requirements change, the parts of a design affected by a given specification change, and the parts of the code affected by a given design change are also important areas for research.

When changing a software system, it is often necessary to reverse an earlier design decision while preserving the later ones. Automated construction and application of meaning-preserving transformations that accomplish this is an important research problem.

3. Conclusion

Advances in software analysis and testing are essential for realizing trusted software systems. Work in this area should be expanded beyond the traditional domain of testing code in a programming language to include software products at all stages of development, from requirements analysis to system evolution. Further work on testing at the code level is also needed, to enable firm conclusions to be drawn from finite sets of test cases constructed by definite and effective methods. Software analysis techniques addressing properties of parallel, distributed, real-time, and knowledge-based systems should be explored as well as those for sequential systems. Well founded and mechanizable analysis techniques are needed for meta-programming and program transformation systems as well as for individual software products.

DISTRIBUTION LIST

(1) Defense Technical Information Center 2
Cameron Station
Alexandria, Virginia 22304-6145

(2) Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Monterey, CA 93943

(3) Center for Naval Analysis 1
4401 Ford Avenue
Alexandria, VA 22302-0268

(4) Director of Research Administration 1
Attn: Prof. Howard
Code 012
Naval Postgraduate School
Monterey, CA 93943

(5) Chairman, Code 52 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

(6) Chief of Naval Research 1
800 N. Quincy Street
Arlington, Virginia 22217

(7) Naval Postgraduate School 100
Code 52Lq
Computer Science Department
Monterey, CA 93943

SOFTWARE EVOLUTION VIA PROTOTYPING

Luqi

Software Evolution via Prototyping

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

Rapid prototyping is widely accepted as an alternative methodology for software development. The problems of software maintenance are magnified in rapid prototyping because prototypes are subject to frequent and repeated changes. The concepts and mechanisms presented in this paper support such changes in rapid prototyping based on component specifications. We discuss the following important issues for software evolution via prototyping: (1) explicit interactions between prototype components for easily determining the impact of a proposed change, (2) requirements tracing facilities for identifying the parts of a prototype affected by a proposed requirements change, (3) structured system construction by maximizing reusability of software components, and (4) the use of specifications in retrieving, composing, and adapting reusable components in minimizing effort for code analysis and modification in software maintenance.

1. Introduction

Software evolution is a most important issue in software development because changes to existing systems account for more than half of the total software cost. Evolution corresponds to the maintenance phase in traditional software life cycle, and consists of the process of firming up the requirements and adapting the software system to maintain the correspondence between the two. It is difficult to eliminate this expensive process because later phases of development often bring more knowledge and more insight into the problem domain and the properties of the intended system to the designers and customers, but the need for some types of changes can be reduced [12]. Software systems are changed for the following reasons:

Requirements errors

The developers have incorrectly understood the requirements and have produced a system that does not meet user needs.

Implementation errors

A faulty design or implementation does not correspond to the specification.

Phased delivery

A partial implementation has been delivered because the customer cannot wait until a complete implementation is available.

User education

Customer requirements have changed because experience with the current version of the system has changed their perception of how computers can be used to solve their problems.

New situations

Changes in the environment of the system have introduced new requirements. Examples of such changes are new external systems, new policies, new technologies, and new competitive pressures.

Prototyping can help reduce the need for unplanned changes by stabilizing the requirements before a significant amount of effort has been invested in implementation. A key problem in large scale software development is the need for communication between people with different areas of expertise. Typically customers know much more about the problem domain than they do about programming, while the programmers know much more about programming than they do about the customer's problems. Both the problem domain and the programming domain have many specialized concepts and terms, many of which are unfamiliar to people who are not experts on the domain. The requirements for the proposed software system are influenced by constraints from both the problem domain and the programming domain that cannot be completely understood without knowledge of specialized concepts from both domains. Requirements are difficult to construct and validate because usually there is no single person who understands all of the constraints on the proposed system. This is especially evident in large systems with hard real-time constraints, since the requirements for such systems are generally very difficult to understand or describe.

A prototype is a concrete executable model of selected aspects of the proposed system. Prototypes are valuable aids in requirements analysis because they can be used to demonstrate the behavior of the proposed system in a form that can be readily understood by all concerned parties. Prototypes can help customers visualize and test consequences of their requirements and provide an effective basis for communication between the customers and the requirements analysts. The process of constructing a prototype also helps the analysts to determine what questions they need to ask to construct a conceptual model of the problem domain that is sufficiently complete to be used in designing the proposed system.

Prototyping can help reduce maintenance costs primarily by reducing requirements errors, so that fewer changes to the software are needed. This applies both to the original formulation of the system and for evolutionary changes sparked by user education or new situations. Phased delivery has traditional been the main mechanism for inducing requirements changes due to user education. Extensive exercising of a prototype by a group of users can trigger some of the requirements changes due to this effect before the production version of the system has been produced. This can alleviate wasted design effort, which is one of the main problems with phased delivery. Phased delivery is usually accomplished by developing a design for the whole system, and then choosing a subset for implementation and delivery in the first release. If the requirements changes due to user education are severe, the design for the rest of the system can be invalidated before it is ever implemented.

Prototyping can also help reduce implementation errors. The prototype can allow more extensive testing of a system by providing a means for evaluating test results. The output of the production code can be mechanically compared to the corresponding output of the prototype, thus allowing more test cases to be examined without increasing the amount of human effort involved. The prototype can also be used by implementors to resolve questions about the intended behavior of the system in particular cases. This can reduce the incidence of errors caused by programmers making plausible but unfounded assumptions about unspecified cases. Such assumptions are often needed in traditional software development because specifications a.e incomplete and schedule pressures do not allow queries to the customer about every minor detail of system behavior. Implementation errors can also be reduced by systematic development techniques supported by formal methods and automated tools [1].

For large systems, every adjustment to the requirements has to be recorded and incorporated into the system specifications, as well as the architecture and the implementation. A systematic way to reduce the extraordinary effort required for software maintenance is to manage the changes in the system needed to reflect adjustments to the requirements at the specification level rather than the implementation level. We can view these two aspects of software maintenance in a unified way if we can mechanically transform specifications into code. While mechanical transformations from black-box specifications into production quality code is not practical at the current time, computer-aided generation of prototype implementations is both feasible and useful [3].

1.1. The Prototyping Life Cycle

The traditional software life cycle consists of a series of phases which yield runnable software only late in the process. One view of the traditional life cycle is illustrated in Fig. 1. A major problem with the traditional approach is that there is no guarantee the resulting product will reliably solve the customer's problem. Often users will be able to indicate the true requirements only by observing the operation of the system, and the traditional life cycle yields executable programs late in the process, when too much money has already been spent and there is no time left to recover from requirements

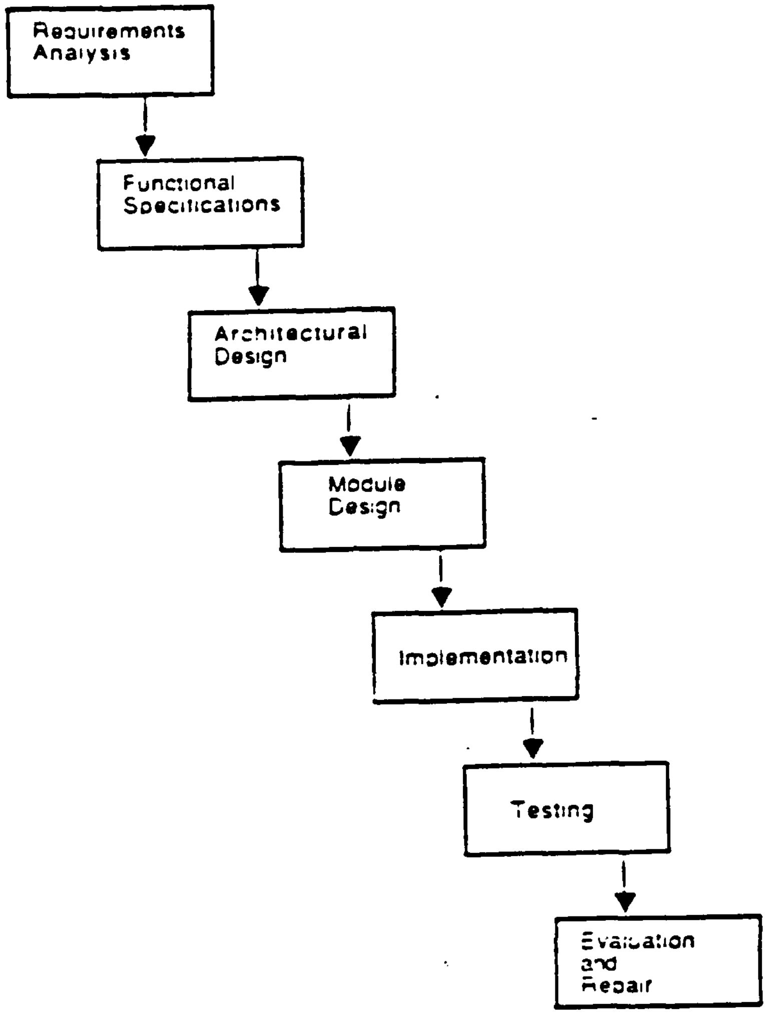


Fig. 1 Traditional Software Life Cycle

errors.

When this traditional life cycle approach is applied to hard real-time or embedded systems, the potential for inconsistencies increases. One of the major differences between a real-time system and a conventional computer system is the required precision and accuracy of the application software. The response time of each individual operation may be a significant aspect of the associated requirements, especially for operations whose purpose is to maintain the state of some external system within a specified region, as is common in embedded software systems. In hard real-time systems response times are a critical determining factor in the accuracy of the software. These response times, or deadlines, must be met or the system will fail to function correctly, with potentially catastrophic consequences. For example, as part of a larger computer system, the requirements for an embedded system can incorporate stringent real-time constraints, parallel processing on multiple computers, and a high degree of reliability. These requirements will often exceed the intellectual capacity of a single software engineer, requiring several individuals working independently on different segments of the system. In such cases the requirements can be very difficult to understand.

Current research suggests a revised software development life cycle, which consists of two phases, rapid prototyping and automatic program generation [8]. This prototyping life cycle is an alternative to the traditional life cycle which has been proposed to alleviate problems stemming from incorrect requirements, especially when designing hard real-time systems. Although current capabilities preclude completely automatic program generation, the required software tools and capabilities do exist for computer-aided rapid prototyping. As a software methodology, rapid prototyping provides the user and designer with a fast, efficient and easy-to-use stepwise process. When utilized during the early stages of the development life cycle, rapid prototyping allows validation of the requirements, specifications and initial design before valuable time and effort are expended on implementation software. Fig. 2 graphically describes this methodology as a feedback loop [8]. Rapid prototyping initially establishes an iterative process between

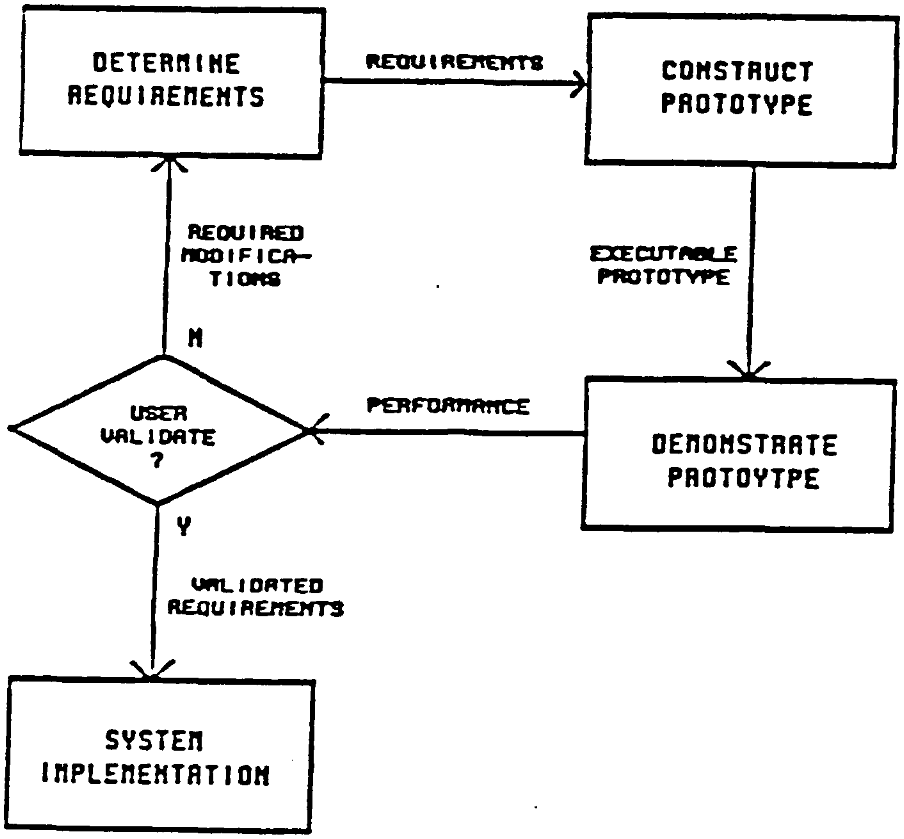


Fig. 2 Prototyping Life Cycle

the user and the designer to concurrently define specifications and requirements for the time critical aspects of the envisioned system. The designer then constructs a model or prototype of the system in a high-level, prototype description language. This prototype is a partial representation of the system, including only those critical attributes necessary for meeting user requirements, and is used as an aid in analysis and design rather than as production software. During demonstrations of the prototype, the user validates the prototype's actual behavior against its expected behavior [9]. If the prototype fails to execute properly or to meet any critical timing constraints, the user identifies required modifications and redefines the critical specifications and requirements. This process continues until the user determines that the prototype successfully meets the time critical aspects of the envisioned system. Following this validation, the designer uses the validated requirements as a basis for the design of the production software.

Computer-aided rapid prototyping further refines the efficiency and accuracy of this new methodology. While utilizing the same iterative approach, computer-aided rapid prototyping relies on software tools which assist the designer in constructing and executing the prototype. We have designed a computer-aided prototyping system (CAPS) to provide an integrated set of tools to support prototyping of complex software systems which may include hard real-time constraints [7]. These tools operate on the prototyping language PSDL (Prototype System Description Language) [10]. This language has been designed to support the needs of rapid prototyping by providing a high level description of the system which can be used to demonstrate the behavior of the prototype by means of the above software tools. Since requirements are especially difficult to define for large systems with hard real-time constraints, PSDL has been designed to apply to such systems.

Prototyping is an iterative process which depends on the ability to rapidly adjust the behavior of the prototype based on feedback from the customer or user. The problems of software maintenance are magnified in rapid prototyping because prototypes are subject to frequent and repeated changes. The goal of the prototyping life cycle is to shift a sizable part of the initial maintenance activity from the production software to the prototype. The potential benefits to be gained from prototyping depend critically on the ability to modify the behavior of the prototype with substantially less effort than that required to modify the production software. The purpose of PSDL and the associated software tools is to provide this ability. The mechanisms by which this is achieved are described below.

1.2. Mechanisms to Support Modifications

The prototyping language PSDL approaches the requirement to support frequent design modifications by means of the following subgoals.

Modularity

The language must make it easy for the system designer to create a prototype with a high degree of module independence and to preserve its good modularity properties across many modifications.

Simplicity

The language should be simple and easy to use.

Reuse

The language should be suitable for specifying the retrieval of reusable modules from a software base.

Adaptability

The language should support small modifications to the behavior of a module without the need to examine its implementation.

Abstraction

The language should support a set of abstractions suitable for describing complex software systems with real-time constraints.

Traceability

The language should support requirements tracing.

Good modularity is essential for achieving ease of modification. An experimental study shows that many of the problems with correctly performing software modifications are due to interactions between widely separated pieces of code [4]. Locality of information was an important design goal of PSDL. The underlying computational model was chosen to make all interactions between components explicit. This model supports a system decomposition criterion that combines data flow and control flow considerations [8].

PSDL is simple and easy to use because it contains a small number of powerful constructs. Designs are described in PSDL as networks of operators connected by data streams. Such networks can be represented as dataflow diagrams augmented with timing and control constraints. The user interface uses the diagrams to provide a convenient means for presenting the system structure to the designer. The operators in the network can be either functions or state machines. The data streams can carry exception conditions in addition to values of arbitrary abstract data types.

PSDL supports reusable components by means of black-box specifications suitable for retrieving modules from a software base. The specification part of a PSDL component contains several attributes which describe the interface and behavior of the component. These attributes can be used to automatically generate a uniform specification for the reusable component [6]. These uniform specifications are used both for retrieval of reusable components and for organizing the software base.

PSDL supports small modifications to modules by means of control constraints. Control constraints can be used to impose preconditions on the execution of a module, to add filters to the output of a module, to suppress or raise exceptions in specified conditions, and to control timers. These facilities allow small modifications to the behavior of a module which do not involve the internal implementation of the module.

PSDL provides abstractions suitable for describing large systems and real-time constraints. These include the non-procedural control constraints mentioned above, timing constraints, timers, functional abstractions, and data abstractions. Timing constraints can be used to associate hard real-time constraints with operators. Examples of timing constraints include the maximum execution time, the maximum response time, and the *minimum calling period*. Timing constraints implicitly determine when operators with hard real-time constraints will be executed. This simplifies the designer's view of the prototype by removing explicit scheduling considerations from tne design of the prototype system. Timers are used to control aspects of behavior that depend on the duration of particular system states or classes of system states. Timers allow static descriptions of durational timing constraints, allowing the designer to ignore the operational details involved in their implementation. A rich set of functional and data abstractions are provided by the pre-defined part of the software base. The designer can define additional functional and data abstractions, either by adding them to the software base or by defining them in terms of more primitive abstractions using PSDL.

PSDL supports requirements tracing by means of a construct for declaring the requirements associated with each part of the prototype. Requirements tracing is important because the prototype must be adapted to the changing perceptions of the requirements resulting from demonstrations of prototype behavior. The links between each requirement and the parts of the prototype realizing the requirement are used to determine which parts of the prototype must be modified when a requirement is changed or dropped. In order to prevent the structure of the design from being corrupted by multiple modifications, it is important to remove parts of the code that are no longer supported by an updated set of requirements. This cannot be done safely unless the correspondence between the requirements and the code is recorded and kept up to date. In situations where this correspondence is not maintained, each change to the system results in the addition of new code, without the removal of any old code. Such a process leads to increasingly complex systems that eventually escape from human control, making removal of old code essential for systems that will be changed many times. The facilities for recording requirements trace information in PSDL are used by software tools in CAPS to provide automated aid in maintaining and using this information.

1.3. Benefits of Prototyping

Prototyping allows an appreciable part of the maintenance activity to be carried out in terms of the prototype rather than in terms of the production code for the intended system. This is useful because the prototype description is significantly simpler than the production code, is expressed in a notation tailored to support modifications, and the software tools in the computer-aided prototyping environment can be used to help carry out the required modifications.

Rapid prototyping is also a useful tool in feasibility studies, for reducing project risks and estimating costs. Prototypes of critical subsystems or difficult parts of a complicated system can significantly increase the confidence that the system can be built before large amounts of effort and expense are committed to the project. Rapid prototyping helps in estimating costs, since the cost of the intended system is usually proportional to the cost of the prototype. The experiences gained in applying rapid prototyping to special applications, eg. database design, the metaprogramming method and others, have substantiated the expected cost relationships between the prototype and the completed system [5]. A prototype can also be used to specify a well modularized skeleton design for the intended system and to validate the important attributes of the intended system, eg. timing constraints, input and output formats, or interfaces between modules.

2. Computer-Aided Prototyping System

The main software tools in the computer-aided prototyping system are shown in Fig. 3. These tools communicate by means of the PSDL language, which serves to integrate the tools and provide a uniform conceptual framework for the prototype designer.

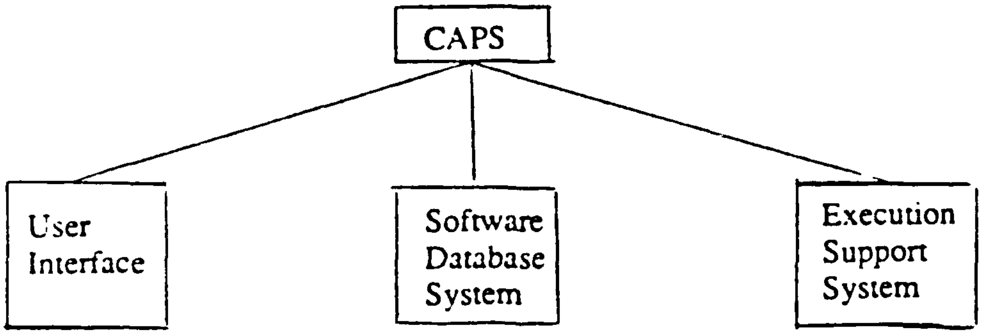


Fig. 3 Main Software Tools in CAPS

The user interface consists of a syntax-directed editor with graphics capabilities, an expert system for communicating with end-users, a browser, and a debugger. The editor enables convenient entry of PSDL descriptions into the system while preventing syntax errors. It also provides support for displaying graphical summary views of the prototype, maintaining the requirements trace, and locating parts of the prototype design related to particular requirements or data streams. The expert system provides a paraphrasing capability which generates English text from PSDL descriptions, to allow end-users to directly examine the prototype without the need for familiarity with PSDL. The browser allows the designer to interact with the software database. In particular the browser provides facilities for retrieving and examining reusable components stored in the software database system. The debugger allows the designer to interact with the execution support system. In particular, the debugger provides facilities for initiating execution of the prototype, displaying results or trace information, and gathering statistics about prototype behavior and performance.

The software database system consists of a design database, a software base, a software design-management system, and a rewrite subsystem. The design database contains the PSDL prototype descriptions for each software development project using CAPS. The software base contains PSDL descriptions and code for all available reusable software components. The software design-management system is responsible for managing and retrieving the versions, refinements, and alternatives of the prototypes in the design database and the reusable components in the software base. The rewrite subsystem translates PSDL specifications into a normalized form that is used by the design-management system for retrieving reusable components from the software base [6].

The PSDL execution support system contains a translator, static scheduler, and a dynamic scheduler [9]. The translator generates code binding together the reusable components extracted from the software base. Its main functions are to implement data streams, control constraints, and timers. The static scheduler allocates time slots for operators with real-time constraints before execution begins. If the allocation succeeds, all operators are guaranteed to meet their deadlines even with worst-case execution times. The dynamic scheduler invokes operators without real-time constraints in the time slots not used by the operators with real-time constraints as execution proceeds.

3. Using CAPS for Maintenance

If the prototyping process is carried out manually, the associated benefits are limited because it takes too much effort. CAPS can increase the leverage of the prototyping strategy by reducing the effort that must be spent by the designer in producing and adapting a prototype to perceived user needs. This section describes how the facilities provided by CAPS can be used to assist in the maintenance activities involved in the prototyping process.

In the prototyping life cycle shown in Fig. 2, the maintenance activity for the prototype starts after the cycle has been carried out once: the analysts have determined the initial requirements by talking to the customers, constructed an initial prototype, and demonstrated it to the customer, who finds some aspects of the prototype's behavior unacceptable and requests some modifications. The process of demonstrating the prototype is aided by the user interface, which has facilities for presenting the results of prototype execution to the customer and for guiding the choice of which aspects of the prototype to demonstrate. The latter function is accomplished by an embedded expert system containing heuristics for exercising prototypes and a facility for recording test case coverage information. The analysts use customer feedback about the behavior of the prototype to modify or refine the requirements, which are maintained as a tree of subgoals. The incremental change leading to the new version of the requirements is entered into the system. At this point the facilities provided by CAPS are used to adapt the prototype to the new requirements.

The user interface helps the prototype design team identify the tasks that have to be carried out to update the prototype. The user interface maintains a list of unresolved new requirements and a list of unresolved modified requirements. Whenever a member of the design team is ready for a new task, the system presents the lists and lets the designer pick an item to resolve. If a modified requirement is chosen, the interface returns a list of modules previously supporting the requirement, and lets the designer check them off as they are adapted or determined to be still valid. The effort required to do this task coordination is minimized by presenting the lists as menus, and allowing the designer to pick items by means of a pointing device. Choosing an item results in a summary view of the module, which can be browsed and updated as required.

The user interface speeds up the process of adapting the prototype by

(1) Helping to coordinate tasks performed by a team of designers,

(2) Helping to focus the designer's attention on the information relevant to a task,

(3) Providing summary views of the system or selected components, and

(4) Locating all potentially relevant parts of the prototype.

The components of the software database actively contribute to the process of adapting the prototype to new requirements. The software design-management system helps to maintain the design history and to locate relevant reusable software components. The design history consists of the relationship between each version of the requirements and the corresponding versions of parts of the prototype. This information is useful because sometimes the customer will retreat to previous versions of the requirements. Situations in which this may happen include cases where the customer gives up on an ambitious requirement in response to cost or performance estimates resulting from examination of the prototype. In such cases parts of the requirements are returned to previous configurations, and the system can help to restore the corresponding parts of the prototype to their previous configurations.

The design database also provides concurrency control functions which allow multiple designers to update the parts of the prototype without risk of unintentional interference. In the interests of minimizing delay, the design database will not lock out access to any part of the design, even while the design is being updated. Instead, the system will allow the previous version of the component to be examined, with a warning that a new version is currently in preparation. The system will provide information about the reason the component is being modified (i.e. some particular new or modified requirement) on request.

The software base provides reusable software components matching given PSDL specifications. In the PSDL prototyping method [8] modules are realized by three main mechanisms:

(1) Retrieval of a suitable component from the software base. The software base should contain flexible generic modules, whose parameters are determined as part of the retrieval process. It also should contain rules for matching a

specification by means of a composite operator, which is realized by a network of operators, at least one of which must be an available reusable component [11]. The retrieval mechanism is therefore capable of performing some routine aspects of bottom-up design, freeing the designer from the need to be familiar with all of the reusable components in the software base.

(2) Decomposition of the component into a network of simpler components. This is done by the designer if the component cannot be retrieved directly from the software base, and the component is sufficiently complex to have a useful decomposition into simpler parts. The designer is responsible for top-down design activities such as inventing new abstractions. Each of the identified parts is specified in PSDL and realized by the same set of mechanisms, applied recursively.

(3) Direct implementation in a programming language. This is done by the designer if the component cannot be retrieved directly from the software base, and the component does not have a useful decomposition into simpler parts. This should be infrequent if the software base is mature, containing the results of prototyping many other systems in the same problem domain.

The execution support system helps to speed up design changes by providing a localized view of the processes in the prototype and by analyzing its timing properties. These features are especially important for prototyping real-time systems. At the programming language level, implementations of real-time systems are difficult to understand because the instructions of several logically independent processes must often be interleaved to meet the timing constraints [2]. PSDL presents a view to the designer in which logically distinct processes are represented as separate independent components. The PSDL execution support system contains a translator which mechanically transfon.ıs this independent representation into the corresponding programming language representation, adding the necessary interleaving in a fashion transparent to the designer.

The timing properties of a real-time system are analyzed by the static scheduler. If the static scheduler succeeds in constructing a schedule, then the operators in the schedule are guaranteed to meet their timing constraints even under worst case operating conditions. In case the static scheduler fails to find a valid schedule, it provides diagnostic information useful for determining the cause of the difficulty and whether or not the difficulty can be resolved by adding more processors [9]. These functions are important because the timing constraints in complex systems can have complicated interactions that can be very difficult to analyze manually.

The prototyping language PSDL is the vehicle for carrying a powerful set of concepts useful for modeling complex systems and for providing a uniform framework for representing prototypes and software components which is common to all of the tools. PSDL also helps the designer achieve good modularity and allows the descriptions of small modifications to component behavior to be separated from the potentially complex implementations of those components.

Good modularity means the prototype should be realized by a set of independent modules with narrow and explicitly specified interfaces. PSDL supports this concept via operators and data streams. An important property of the language is that two distinct operators can communicate or affect each other's behavior only by means of the data streams explicitly connecting them, either directly or indirectly. This locality property is important for maintenance because it allows the set of modules that can potentially interact with a given module to be determined via a simple mechanical analysis of the data flow network, allowing the software tools to guarantee that all aspects of a proposed change have been covered. It also encourages designs containing an independent component for each major design decision. Such designs are easier to modify because the information required to change a design decision is localized in one region of the code.

The locality property is embodied by the PSDL scoping rules and mechanically enforced. The implementation of an operator can only refer to the explicitly declared input and output streams of the operator and to data streams local to the implementation of the operator. Implementations of operators representing state machines may contain closed loops consisting of local data streams.

PSDL supports small modifications to the behavior of a component by means of control constraints. This mechanism can be used to adapt the behavior of a prototype to make a small change without examining the internal implementation of the affected component. For example, a common kind of problem discovered in the demonstration of a prototype is that a given operator has the intended behavior most of the time but not always. The PSDL control constraints governing conditional execution of operators are useful in such a case. A control constraint in the form of an input guard predicate can be added in such a case, where the guard predicate describes the circumstances in which the execution of the operator will produce the intended result, and serves to disable the execution of the operator in cases where it would not produce the correct result. This allows the addition of another operator for producing the correct output in the remaining cases, controlled by a complementary guard predicate. Another example is a case where an operator is discovered to have an inappropriate response to ill-formed inputs. The PSDL control constraints governing conditional outputs or exceptions are useful in such a case. Such control constraints introduces an output guard predicate, which serves to disable the output of the operator if it does not satisfy the predicate, or an exception triggering predicate, which produces an exception instead of the value computed by the operator. Output guard predicates can be used to filter out inappropriate responses in cases where no response is needed. In particular, they can be used to disable exceptions raised by implementations of components if the conditions reported by the exception do not require any action on the part of the prototype. Exception triggering predicates can be used to trigger exceptions when incorrect outputs have been computed, or to rename exceptions produced by the implementation of a module to other conditions meaningful at a higher level. Triggering an exception is useful because it allows a new module to be added for handling the exception, again without affecting the implementation of the original component. Renaming exceptions is useful for repairing inappropriate error messages. For example, in the context of [illegible] operating system, a process_table_overflow condition might be translated into a "machine_busy" condition to convert an internal view of a failure in terms of the implementation to an external view meaningful to the users of the prototyped system. An exception triggering predicate is used instead of an input guard predicate in cases where the condition to be checked depends on the output values of the operator in addition to its inputs values.

4. Conclusions

The effort required for supporting the evolution of a software system can be reduced via prototyping. Prototyping can be used to stabilize the requirements for either an initial software development project or a proposed enhancement to an existing system. Especially for systems with complex requirements, such as large or embedded real-time systems, human communication is ineffective without the feedback provided by demonstrations of proposed system behavior. An iterative process involving modifications to perceived requirements and proposed system behavior is needed to arrive at a common understanding of the proposed system by the customer and the developer. It is more cost effective to use a prototype rather than production quality code to provide the demonstrations of proposed system behavior because prototypes are simpler and easier to modify than production quality implementations.

The effectiveness of prototyping is limited if it must be carried out manually. A high level language, a systematic prototyping method, and an integrated set of computer-aided prototyping tools are important for realizing the potential benefits of prototyping. The prototyping effort is also aided by a powerful set of abstractions appropriate for a problem domain, especially if these abstractions are embodied in a set of reusable software components. Effort can be saved in the long run by building up a comprehensive library of such components for an application area, especially if more than one software system must be developed for the same problem domain, which is often the case.

A typical software system evolves in a long series of repairs and enhancements, in response to the discovery of faults and to changes in user requirements. Most useful systems are too large to be maintained by just one person, leading to the need for coordinating the concurrent efforts of a group of people. Enhancements to software systems must often be developed concurrently even though they are not independent. Sometimes the designers talk to each other as they proceed, and adjust their designs to make sure they are compatible. In other cases, the designs are developed independently, and then are merged at the end, with both designers examining each other's code and making adjustments as needed. Manual methods for merging enhancements are inadequate because they are slow and error prone. Automatic methods for merging enhancements are needed. To be useful, such methods should provide some assurances that the results are correct, or else locate potential inconsistencies.

A better understanding of the software merging problem and better computer aided design tools enabled by that understanding are important because the bulk of the cost of a software system is due to enhancements and repairs. Coordinating the efforts of many people working on the same software system is difficult and expensive, and a design style that allows people to work more independently and uses automatic merging of independent enhancements should reduce communication and coordination problems.

The ability to easily swap in different alternative choices for an aspect of the behavior of a system would also be useful in using a prototype to aid in determining user requirements. Automated merging would make it practical to customize software products for each user's needs by picking options from a multiple choice menu, as is commonly done for automotive products now. Such an approach would make software systems more flexible and make it less critical to get the requirements right the first time. It would also make it easier to design and maintain a family of software products intended to exist in a variety of configurations.

1. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1988.

2. S. Faulk and D. Parnas, "On Synchronization in Hard-Real-Time Systems", *Comm. of the ACM 31*, 3 (Mar. 1988), 274-187.

3. P. Freeman, "A Conceptual Analysis of the Draco Approach to Constructing Software Systems", *IEEE Trans. on Software Eng. SE-13*, 7 (July 1987), 830-844.

4. S. Letovsky and E. Soloway, "Delocalized Plans and Program Comprehension", *IEEE Software 3*, 3 (May 1986), 41-49.

5. L. Levy, "A Metaprogramming Method and Its Economic Justification", *IEEE Trans. on Software Eng. SE-12*, 2 (Feb. 1986), 272-277.

6. Luqi, *Normalized Specifications for Identifying Reusable Software*, Proc. of the ACM-IEEE 1987 Fall Joint Computer Conference, Dallas, Texas, October 1987.

7. Luqi and M. Ketabchi, "A Computer Aided Prototyping System", *IEEE Software 5*, 2 (March 1988), 66-72.

8. Luqi and V. Berzins, "Rapidly Prototyping Real-Time Systems", *IEEE Software*, Sep. 1988, 25-36.

9. Luqi and V. Berzins, "Execution of a High Level Real-Time Language", in *Proc. of the Real-Time Systems Symposium*, Dec. 1988.

10. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. on Software Eng.*, October, 1988.

11. Luqi, "Knowledge Base Support for Rapid Prototyping", *IEEE Expert*, Nov. 1988.

12. R. Yeh and T. Welch, "Software Evolution: Forging a Paradigm", in *Proc. Fall Joint Computer Conference*, ACM and IEEE Computer Society, Oct. 1987, 10-12.

Initial Distribution List

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22314

Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Monterey, CA 93943

Center for Naval Analysis 1
4401 Ford Avenue
Alexandria, VA 22302-0268

Office of the Chief of Naval Operations 2
Code OP-941
Washington, D.C. 20350

Office of the Chief of Naval Operations 2
Code OP-945
Washington, D.C. 20340

Commander Naval Telecommunications Command 2
Naval Telecommunications Command Headquarters
4401 Massachusetts Avenue NW
Washington, D.C. 20390-5290

Commander Naval Data Automation Command 1
Washington Navy Yard
Washington, D.C. 20374-1662

Office of Naval Research 1
Office of the Chief of Naval Research
Attn. CDR Michael Gehl, Code 1224
Arlington, VA 22217-5000

Director, Naval Telecommunications System Integration Center 1
NAVCOMMUNIT Washington
Washington, D.C. 20363-5100

Space and Naval Warfare Systems Command 1
Attn: Dr. Knudsen, Code PD50
Washington, D.C. 20363-5100

Ada Joint Program Office 1
OUSDRE(R&AT)
The Pentagon
Washington, D.C. 230301

Naval Sea Systems Command 1
Attn: CAPT Joel Crandall
National Center #2, Suite 7N06
Washington, D.C. 22202

Office of the Secretary of Defense 1
Attn: CDR Barber
The Star Program
Washington, D.C. 20301

Naval Ocean Systems Center 1
Attn: Linwood Sutton, Code 423
San Diego, CA 92152-5000

National Science Foundation 1
Division of Computer and Computation Research
Washington, D.C. 20550

Director of Research Administration 1
Code 012
Naval Postgraduate School
Monterey, CA 93943

Chairman, Code 52 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

LuQi 150
Code 52Lq
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

SOFTWARE ANALYSIS AND TESTING THROUGH PROTOTYPING

Luqi
B. Kramer
V. Berzins

Software Analysis and Testing through Prototyping

Luqi
B. Kraemer
V. Berzins

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

Prototyping is such a complementary approach, which allows many of the traditional kinds of software analysis and testing to be applied at earlier stages. The prototyping process helps to establish relatively static concepts of correctness, which can be used as a meaningful basis for later verification efforts. The execution of software prototypes is similar to traditional validation, except that the developer is explicitly concerned with the length of time during which the proposed system will continue to meet customer needs, rather than just ensuring the system will meet currently perceived needs. This paper also discusses three levels of analysis and testing that are important for real-time systems in rapid prototyping.

1. Introduction

Robustness, reliability, and correctness of operation are quality aspects of software products that gain increasing importance. This is particularly true for critical systems whose malfunction may result in loss of human life, compromise of national security, or massive loss of property [3]. Software components of hard real-time systems often are among this class of critical system as they typically control production processes, transport systems, communication systems, chemical or power plants, etc.

The techniques for certifying such properties range from formal program *verification* and software *testing* to formal and informal *analysis* techniques [2,5] such as data flow analysis, loop invariant detection, deadlock detection, software inspection, documentation evaluation, and walkthroughs. Overview information and references to verification can be found in *Software Engineering Notes*, August 1985, information about current approaches to testing are surveyed in [4], and other validation techniques are reported in [1] and *IEEE Software*, May 1989. All of these techniques have their specific merits but also show limitations which require using a combination of validation and verification techniques for building quality software.

Testing has a long tradition in programming and software engineering. The traditional approach to software development, however, suffers from the fact that the results of testing operational code become available close to the end of the development process, so that design errors detected during system testing require an immense redesign and reimplementation effort, and are likely to cause project delays if they occur. Thus an effective quality assurance strategy should combine testing with other approaches that can detect requirements and design errors earlier in the cycle, when they are less expensive to correct and have less external impact on the project. Prototyping is such a complementary approach, which allows many of the the traditional kinds of software testing to be applied at earlier stages.

2. The Role of Prototyping in System Validation

The faults in software systems with the largest impact are requirements and specification errors, since such errors tend to affect large portions of the system and can be very expensive to correct. Requirements are often uncertain at the early stages, because the customers do not have a complete understanding of their problems or how proposed software systems will affect their daily operations and their understanding of the application domain. Experience with a software system usually changes the customers' perceptions of their problems, and opens up new possibilities which lead to new requirements. This makes the customer's problem a *moving target*. Educating customers and developers about the problem is as much a part of the process as building a system, so that the traditional concepts of what constitutes an error do not quite match the reality of the early parts of the development process: a system that is "correct" at one point in time may become "incorrect" later without any changes in system behavior.

The purpose of the iterative prototyping process illustrated below is to help stabilize the effects of a proposed system on the customer's perceptions and requirements before the system is constructed on a full scale.

```
+-------------+                     +-----------+
| determine   |   requirements      | construct |
| requirements|-------------------->| prototype |
+-------------+                     +-----------+
      ^                                   |
      | requirements adjustment           | prototype
      |                                   |
+-------------+                           v  +----------------+
| demonstrate |<--------------------------+-->| system         |
| prototype   |------------------------------>| implementation |
+-------------+        requirements OK        +----------------+
```

This process helps to establish relatively static concepts of correctness, which can be used as a meaningful basis for later verification efforts. The goal of prototyping is similar to traditional validation, except that the developer is explicitly concerned with the length of time during which the proposed system will continue to meet customer needs, rather than just ensuring the system will meet currently perceived needs. Automated tools are necessary to carry out this process with reasonable speed and cost [1,6].

3. Multiple Levels of Analysis and Testing

There are at least three levels of analysis and testing that are important for real-time systems at the prototyping stage:

(1) Checking whether proposed timing requirements are sufficient to meet the higher level functional requirements that motivate the timing requirements. Common examples of such functional requirements include ensuring that software estimates of the state of a real world sys'em are maintained to a given accuracy, or that the state of the real-world system is controlled to remain within some desirable region. A concrete example is an aircraft control system, whose purpose is to prevent mid-air collisions. Testing is essential for this part of the problem, because it involves the relationship between a formally described abstract object (the software prototype) and an informally described concrete object (the physical system to be controlled). Analysis of formal models of the software and interacting physical systems should be coupled with testing to check the correspondence between the formal models and the real world.

(2) Checking whether a proposed design meets its requirements, given that the individual components meet their specifications. Most large software systems are designed using modular decompositions. The essential question at the design stage is: will a proposed design work correctly if the implementations of the specified subcomponents are carried out with perfect accuracy? A specification-based prototyping approach can help answer this question before much effort has been spent on the detailed implementation of the components. This part of the problem is subject to formal verification techniques, which are easier than prov-

ing correctness of low level code, because only the correspondence between two sets of specifications in the same language is at issue.

(3) Checking whether a component meets its specifications and timing constraints. This process can be addressed at the prototyping stage by testing and instrumentation that monitors the behavior of an executing prototype with respect to its specifications. This part of the problem has both symbolic and testing aspects, particularly with respect to the real-time behavior of the proposed implementation, which again depends to some extent on the physical properties of the hardware systems involved in the implementation. To establish some confidence that a proposed system will provide guaranteed service within a deadline, the interactions between the software with users, hardware, and other physical components must be tested.

4. How It Can Be Done

We propose a rapid prototyping approach comprising a language, called RPL, [8] and an integrated tool set supporting iterative prototyping of complex software systems [6,7,10]. RPL covers a wide range of applications, including real-time, parallel, distributed, and knowledge-based systems. It combines second-order logic specifications supporting verification with an augmented dataflow representation for design and interconnection of prototype components. The design graph is augmented with special pre/post conditions to express real-time constraints and adjust component behavior to each application context [9]. Execution is based on automatically generating code which links reusable software components or simulates component behavior via an executable subset of the specification logic. Real-time constraints are guaranteed by automatically constructed schedules. Iterative modifications of prototypes are supported by localized information in RPL and its computational model, component behavior modification via logical constraints, and facilities of the tool set for code and design reuse, requirements tracing, and static analysis. The logic and the proposed computational model provide the basis for integrating these facilities into a coherent language and tool structure. Among others the tool set will support execution and dynamic debugging, optimization and transformation to final implementation, as well as formal analysis and proofs of correctness.

The prototyping approach allows requirements and desirable features of the intended system to be clarified while the system is incrementally implemented by mapping designs to reusable and executable software components. Design alternatives can be evaluated by observing the behavior of prototypes under real-time conditions. Test data generation is simplified due to the the separation of concerns emphasized by RPL and its formal semantics. Predicted performance can be verified by executing the prototype under real-time conditions reflecting best and worst case assumptions. In particular, static analysis can be combined with testing to verify the assumptions of the timing properties of the software components on which the design is based, with special attention to the paths with the longest expected execution times. This can lead to greater confidence by decoupling the empirical estimation of the execution times for individual machine instructions from the static analysis which determines the sequence of instructions along the longest execution path.

5. Conclusion

The interaction between testing and prototyping should be explored from several points of view. Since prototypes are embedded in a computer-aided prototyping system for execution, they provide a greater degree of flexibility, observability, and control than a production implementation, enabling new testing techniques that check some of the critical decisions made in the early stages of software development, and provide a means for coupling testing with simplified formal analysis with respect to high level specifications. As in Monte Carlo simulations, the use of partial formal analysis to reduce the variability of the unknown aspects of the problem can lead to more accurate conclusions based on fewer test cases. Demonstrations to customers also provide a means for using testing techniques to do requirements validation, and provide error detection and location earlier in the development process, when it can have a much larger beneficial effect. These possibilities open up a new and important area for future research and development.

1. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1989.

2. M. Christ-Neumann, B. Kraemer, H. H. Nieters and H. W. Schmidt, "The GRASPIN Environment on the Lisp Machine - User's Guide", GRASPIN Technical Paper GMD37/1, GMD, Sankt Augustin, Mar 1989.

3. J. Goguen, "OBJ as a Theorem Prover with Applications to Hardware Verification", SRI-CSL-88-4R2, SRI International, Menlo Park, California, August 1988 .

4. W. Howden, *Functional Program Testing and Analysis*, McGraw-Hill, New York, 1987.

5. B. Kraemer, *Concepts, Syntax and Semantics of SEGRAS - A Specification Language for Distributed Systems*, Oldenbourg Verlag,, Muenchen-Wien, 1989.

6. Luqi, ''Rapid Prototyping for Large Software System Design'', Ph. D. Thesis, University of Minnesota, 1986.

7. Luqi and V. Berzins, ''Rapidly Prototyping Real-Time Systems'', *IEEE Software*, Sep. 1988, 25-36.

8. Luqi, V. Berzins, B. Kraemer and L. White, ''A Proposed Design for a Rapid Prototyping Language'', NPS52-89-45, Computer Science Department, Naval Postgraduate School, 1989.

9. Luqi, ''Handling Timing Constraints in Rapid Prototyping'', in *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, Jan. 1989, 417-424.

10. W. Swartout and R. Balzer, ''On The Inevitable Intertwining of Specification and Implementation'', *Comm. of the ACM 25*, 7 (July 1982), 438-440.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
Cameron Station
Alexandria, Virginia 22304-6145

2. Library, Code 0142 2
Naval Postgraduate School
Monterey, California 93943-5002

3. Office of Naval Research 1
Office of the Chief of Naval Research
Attn. CDR Michael Gehl, Code 1224
800 N. Quincy Street
Arlington, Virginia 22217-5000

4. Space and Naval Warfare Systems Command 1
Attn. Dr. Knudsen, Code PD 50
Washington, D.C. 20363-5100

5. Ada Joint Program Office 1
OUSDRE(R&AT)
Pentagon
Washington, D.C. 20301

6. Naval Sea Systems Command 1
Attn. CAPT Joel Crandall
National Center #2, Suite 7N06
Washington, D.C. 22202

7. Office of the Secretary of Defense 1
Attn. CDR Barber
STARS Program Office
Washington, D.C. 20301

8. Office of the Secretary of Defense 1
Attn. Mr. Joel Trimble
STARS Program Office
Washington, D.C. 20301

9. Commanding Officer 1
Naval Research Laboratory
Code 5150
Attn. Dr. Elizabeth Wald
Washington, D.C. 20375-5000

10. Navy Ocean System Center 1
Attn. Linwood Sutton, Code 423
San Diego, California 92152-500

11. National Science Foundation 1
Attn. Dr. William Wulf
Washington, D.C. 20550

12. National Science Foundation 1
Division of Computer and Computation Research
Attn. Tom Keenan
Washington, D.C. 20550

13. National Science Foundation 1
Director, PYI Program
Attn. Dr. C. Tan
Washington, D.C. 20550

14. Office of Naval Research 1
Computer Science Division, Code 1133
Attn. Dr. Van Tilborg
800 N. Quincy Street
Arlington, Virginia 22217-5000

15. Office of Naval Research 1
Applied Mathematics and Computer Science, Code 1211
Attn: Dr. James Smith
800 N. Quincy Street
Arlington, Virginia 22217-5000

16. New Jersey Institute of Technology 1
Computer Science Department
Attn. Dr. Peter Ng
Newark, New Jersey 07102

17. Southern Methodist University 1
Computer Science Department
Attn. Dr. Murat Tanik
Dallas, Texas 75275

18. Editor-in-Chief, IEEE Software 1
Attn. Dr. Ted Lewis
Oregon State University
Computer Science Department
Corvallis, Oregon 97331

19. University of Texas at Austin 1
Computer Science Department
Attn. Dr. Al Mok
Austin, Texas 78712

20. University of Maryland 1
College of Business Management
Tydings Hall, Room 0137
Attn. Dr. Alan Hevner
College Park, Maryland 20742

21. University of California at Berkeley 1
Department of Electrical Engineering and Computer Science
Computer Science Division
Attn. Dr. C.V. Ramamoorthy
Berkeley, California 94720

22. University of California at Los Angeles 1
School of Engineering and Applied Science
Computer Science Department
Attn. Dr. Daniel Berry
Los Angeles, California 90024

23. University of Maryland 1
Computer Science Department
Attn. Dr. Y. H. Chu
College Park, Maryland 20742

24. University of Maryland 1
Computer Science Department
Attn. Dr. N. Roussapoulos
College Park, Maryland 20742

25. Kestrel Institute 1
Attn. Dr. C. Green
1801 Page Mill Road
Palo Alto, California 94304

26. Massachusetts Institute of Technology 1
Department of Electrical Engineering and Computer Science
545 Tech Square
Attn. Dr. B. Liskov
Cambridge, Massachusetts 02139

27. Massachusetts Institute of Technology 1
Department of Electrical Engineering and Computer Science
545 Tech Square
Attn. Dr. J. Guttag
Cambridge, Massachusetts 02139

28. University of Minnesota 1
Computer Science Department
136 Lind Hall
207 Church Street SE
Attn. Dr. Fox
Minneapolis, Minnesota 55455

29. International Software Systems Inc. 1
12710 Research Boulevard, Suite 301
Attn. Dr. R. T. Yeh
Austin, Texas 78759

30. Software Group, MCC 1
9430 Research Boulevard
Attn. Dr. L. Belady
Austin, Texas 78759

31. Carnegie Mellon University 1
Software Engineering Institute
Department of Computer Science
Attn. Dr. Lui Sha
Pittsburgh, Pennsylvania 15260

32. IBM T. J. Watson Research Center 1
Attn. Dr. A. Stoyenko
P.O. Box 704
Yorktown Heights, New York 10598

33. The Ohio State University 1
Department of Computer and Information Science
Attn. Dr. Ming Liu
2036 Neil Ave Mall
Columbus, Ohio 43210-1277

34. University of Illinois 1
Department of Computer Science
Attn. Dr. Jane W. S. Liu
Urbana Champaign, Illinois 61801

35. University of Massachusetts 1
Department of Computer and Information Science
Attn. Dr. John A. Stankovic
Amherst, Massachusetts 01003

36. University of Pittsburgh 1
Department of Computer Science
Attn. Dr. Alfs Berztiss
Pittsburgh, Pennsylvania 15260

37. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. Dr. Jacob Schwartz
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

38. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. Dr. Squires
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

39. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. MAJ Mark Pullen, USAF
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

40. Defense Advanced Research Projects Agency (DARPA) 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, Virginia 2209-2308

41. Defense Advanced Research Projects Agency (DARPA) 1
Director, Strategic Technology Office
1400 Wilson Boulevard
Arlington, Virginia 2209-2308

42. Defense Advanced Research Projects Agency (DARPA) 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, Virginia 2209-2308

43. Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, Virginia 2209-2308

44. MCC AI Laboratory 1
Attn. Dr. Michael Gray
3500 West Balcones Center Drive
Austin, Texas 78759

45. COL C. Cox, USAF 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

46. University of Maryland 1
Attn. Dr. Basili
Computer Science Department
College Park, MD 20742

47. University of California at San Diego 1
Department of Computer Science
Attn. Dr. William Howden
La Jolla, California 92093

48. University of California at Irvine 1
Department of Computer and Information Science
Attn. Dr. Nancy Levenson
Irvine, California 92717

49. University of California at Irvine 1
Department of Computer and Information Science
Attn. Dr. L. Osterweil
Irvine, California 92717

50. University of Colorado at Boulder 1
Department of Computer Science
Attn. Dr. Lloyd Fosdick
Boulder, Colorado 80309-0430

51. Santa Clara University 1
Department of Electrical Engineering and Computer Science
Attn. Dr. M. Ketabchi
Santa Clara, California 95053

52. Oregon Graduate Center 1
Portland (Beaverton)
Attn. Dr. R. Kieburtz
Portland, Oregon 97005

53. Dr. Wolfgang Halang 1
Bayer AG
Ingenieurbereich Progessleittechnik
D-4047
Dormagen, West Germany

54. Dr. Bernd Kraemer 1
GMD Postfach 1240
Schloss Birlinghaven
D-5205
Sankt Augustin 1, West Germany

55. Dr. Aimram Yuhudai 1
Tel Aviv University
School of Mathematical Sciences
Department of Computer Science
Tel Aviv, Israel 69978

56. Dr. Robert M. Balzer 1
USC-Information Sciences Institute
4676 Admiralty Way
Suite 1001
Marina del Ray, California 90292-6695

57. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn. Mr. Samuel A. DiNitto, Jr.
Griffis Air Force Base, New York 13441-5700

58. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn. Mr. William E. Rzepka
Griffis Air Force Base, New York 13441-5700

59
LuQi **100**
Code 52Lq
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

60 Steve Huseth
Honeywell Systems & Research Center 1
3660 Technology Dr
Mels, MN 55418

61 Research Administration
Code: 012 1
Naval Postgraduate School
Monterey, CA 93940

PETRI NET-BASED MODELS OF SOFTWARE ENGINEERING PROCESSES

B. Kramer
Luqi

Petri Net-Based Models of Software Engineering Processes

Bernd Krämer and Luqi
Naval Postgraduate School, Code 52
Monterey, CA 93943
kraemer@nps-cs.arpa

July 11, 1989

Abstract

We present an extension of the classical Petri net model to formally define functional, structural, and dynamic aspects of software engineering processes. In this model Petri nets are augmented with logic specifications that serve to specify the essential static properties of software objects involved in a process and define global constraints to the dynamic behavior of process models. These models have an intuitive, causality-based execution semantics which enables process simulation and formal analysis using tools and techniques that have been developed for a related Petri net-based specification formalism. Structuring mechanisms are provided to support hierarchical decomposition and the systematic combination of separate views of a software engineering process. As an example we model selected aspects of a rapid prototyping process which supports the reuse of archived software components and guides the use of dedicated prototyping tools.

1 Introduction

A criticism of traditional life cycle models has been the subject and motivation of many recent papers arguing for new approaches to software process modeling, e.g., [1, 3, 4, 8]. Rather than paraphrasing their criticism, we restrict ourselves to subsuming evaluation criteria we found in the literature and providing a few supplementary remarks to justify our own approach of a Petri net-based process model (PNP model) and narrow down the range of issues it tackles.

Typical requirements posed to process models are *adequacy* of the model, *readability* and *ease of use*, *hierarchical decomposability*, and amenability to *formal analysis* and *reasoning*. The arguments supporting these requirements are largely obvious, except for the notion of adequacy which is difficult to grasp due to the manifold aspects software engineering processes comprise. They include, for example, management aspects concerning the optimal employment of people and use of material resources, contractual matters, planning and cost issues, communication and synchronization aspects, or methodological concerns aiming at effective development procedures and tool use.

As we can hardly imagine a homogeneous process model capturing all these different aspects in an adequate way, we first discuss in the conceptual framework which the PNP model covers. Basic concepts of the PNP model are described in Section 3. We emphasize a formal approach to specifying the dynamic behavior of software engineering processes and characteristic attributes of software objects and roles of human participants involved. An illustrative example is given in Section 4 where we present two partially overlapping views of a rapid prototyping process that supports evolutionary software development by interactive construction of executable prototypes from reusable software components [12]. In Section 5 we illustrate constructions that allow consistent combinations and stepwise refinements of process model views In Section 6 the Petri net semantics underlying PNP models is sketched and their poter. llow formal analysis and reasoning, verification, and symbolic simulation is outlinc

2 Behavior-Oriented Softwaic Process Models

Software development is a dynamic and distributed activity in which many cooperating participants may act partially independently of each other to iteratively transform an initial set of requirements into a validated object system. Different participants usually have different and selective knowledge about an evolving software system. The object system is typically characterized by a large set of software components such as requirements definitions, design documentation, specification and program modules, test protocols and the like which coexist at designated development states.

In this context model adequacy means to capture the distributedness and combinatorial nature of information characterizing an object system in its various development states and the distributedness of changes it undergoes. Speaking in technical terms, a process model approach must be able to handle behavioral issues such as *concurrency*, *synchronization*, and *communication*. It also means to cope with *nondeterminism* occurring in different forms in the course of a development process. For example resource contention is likely to arise due to the boundedness of resources but often cannot be resolved as a process model is designed; or it might be necessary to specify the range of alternative possibilities to pursue a process execution without being able to provide a deterministic decision procedure becaues it depends on information that cannot be anticipated in sufficient detail.

The dynamic behavior of a process model strongly depends on the structure of software components and specific roles of human participants in that process. Therefore it is crucial to provide abstraction mechanisms that allow the process designer to define *functional* and and *structural* properties of objects and roles at a level of detail that is necessary to understand and control a development process but still admits developers to make design decisions as needs arise.

A suitable abstraction of a program module in the context of version control, for example, might describe its structure as consisting of a name, author, interface, and body attribute. The role of programmers acting as authors of such modules might be sufficiently characterized by access rights determining who is allowed to update which program modules. The behavior of the version control model then would specify at this structural level how and under which conditions these attributes can be changed by processes but would not refer to details of a module body, for instance. These changes include update rights as the team of programmers involved in a project or their roles may change and new modules are constructed as the system evolves.

3 Basic Concepts of the PNP Model

To capture equally well functional, structural, and behavioral aspects of software processes, the PNP model extends the classical Petri net model by object-oriented data abstraction facilities. The latter allow the process designer to introduce different types of software objects and roles of human participants, provide them with distinguishing attributes, and describe functional relations between between them. Petri net concepts serve to adequately specify the rules governing distributed changes to defined attributes and relationships. The combination provides a suitable notion of distributed states and state-dependent actions that can dynamically create new software objects, concurrently change their properties, and delete objects that are no longer needed.

3.1 Static Aspects of Objects

Software objects are treated as typed and uniquely named entities whose structure and properties are expressed in terms of extensible lists of *attributes*. Attributes either are (references to) objects or are data. Data specifications are supported in the PNP model based on typed Horn clause logic similar to the specification approach defined in [10].

New *object types* are introduced by a special form relating a new type name with names and types of attributes which all instances of that object type share. For example, the form

```
object module:  (ext-name,author:name, if:interface, body:impl)
```

defines objects of type **module** to have at least five attributes whose values are of type **name**, **interface**, and **impl** respectively. These attributes might capture those properties of program module relevant for configuration management.

Attribute names like **ext-name**, **author**, **if**, and **body** denote (projection) functions mapping the object type into the corresponding attribute type. Further attributes can be added to an object as needs arise. But they can only be accessed by pattern matching using the following tuple notation for objects:

```
<M,[N,A,I,B,new]>
```

where **new** is such an add-on attribute value.

Similarly to objects, immutable data structures which are composed of a specific list of component data or have a variant type and value can easily be defined using two forms that are inspired by the object-oriented data model introduced in [11]. An example of the first kind is the data structure abstracting from module interfaces as consisting of two lists of facilities that are exported and imported:

```
record interface:  (export,import:[facility])
```

where angle brackets denote a list of items of the type the enclose. An example of the second kind is the following:

```
variant eval-state:  (new,ckd,vd:unit)
```

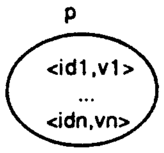
It is a trivial variant data structure which just enumerates a finite set of distinct constants used to denote the evaluation status of a software object.

3.2 Dynamic Aspects of Objects

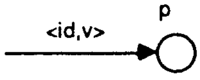
Objects are created dynamically during process execution. Most of the objects created persist as system development proceeds and simply change their attribute values. But there may also be situations in which it is useful to specify that objects are no longer needed and are better discarded. For example, patches to certain program modules can be deleted once a new system version including the dynamically patched changes has been released.

Dynamic object creation, modification, deletion, and changes to mutable relationships among objects and data are captured by ***variable predicates*** whose extension is changed by occurrences of instances of *actions* which schematically specify similar rules of change

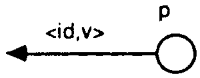
The form denotes



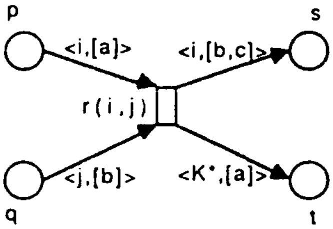
the place of all objects <idi,vi> satisfying the variable predicate p in the present development state



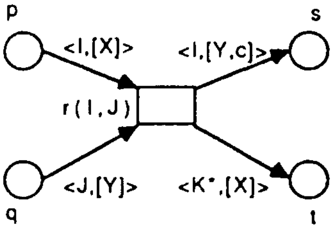
the change element making object <id,v> begin to satisfy p



the change element making object <id,v> cease to satisfy p



a rule of change which is symbolized by the term r(i,j) and consists of several change elements including the creation of an object with identity K and attribute list [a] and the deletion of object <j,[b]>; the labeling of arcs expresses the idea that the lifeline of object K starts and that of object j ends with an application of the given rule



constraining r(I,J) by $X \leq Y$

a scheme of similar rules of change (an action); an instance of an action is obtained by consistently substituting the variables I,J,K,X,Y in the scope of the action by constants such that the formula constraining the action is satisfied according to the specified meaning of functions and predicates it is composed of; note that constants like c express commonalities which all instances share

Notational remarks: Variables are capitalized to distinguish them from function, predicate, and relation names, which are written in lower case.

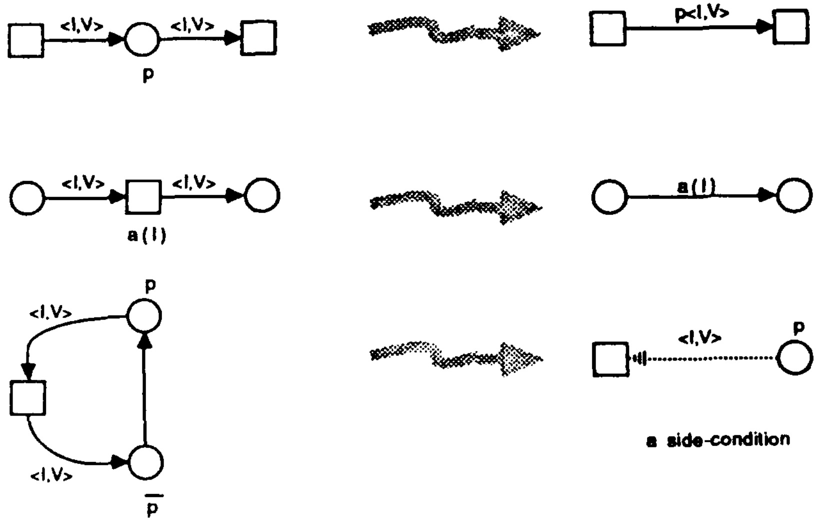
Figure 1: Expressing dynamic aspects cf software engineering processes

in distributed processes. As a simple graphical notation for representing these dynamic aspects of software processes, we use Petri nets that are annotated as shown in Fig. 1.

This notation reinterprets basic concepts of high-level Petri nets (see, e.g. [6, 7]) in a specific way to reflect concepts of the application domain. Objects are always represented as pairs. The first component is a unique identity which is implicitly provided as an object is created. It can never be changed and allows one to follow the lifeline of an object and all changes it underwent. The second component is a list of attribute values given in the order determined by the corresponding object definition. Object creation is made explicit by append a * to the variable referring to a new object. Object deletion is simply expressed by letting the lifeline of an object end in an action. A deleted object is no longer accessible in the further course of a software proccess.

3.3 Example

Using this notation and the following abbreviations



a model of a simple version control system providing only one action to release private modules as substitutes for previous versions kept in a public module library can be composed as shown in Fig. 2. The side-condition of this action requires that only those authors may put their private module into the public library who are assigned the right to update library modules of the proper name.

To keep the example simple, it gives only an incomplete view of our simple version control system. This view does not show how new module names are inserted in the library, how private versions are constructed, and how update rights are modified as module names are created or author names change. As we shall see from later sections, this sort of constructing separate and incomplete views of a process model is supported by

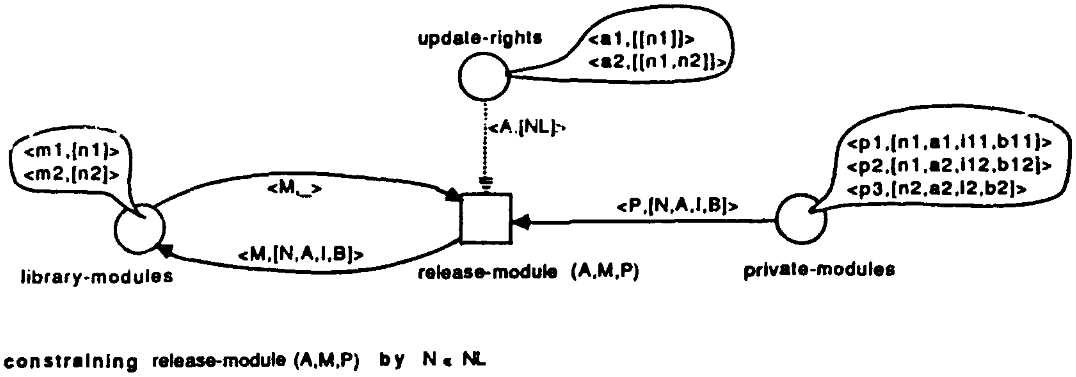


Figure 2: A process model controling the release of private modules as public versions

combination mechanisms that allow one to merge simple views in a consistent way to larger and more complex ones.

3.4 Development States and State Transitions

In a PNP model as shown in Fig. 2 development states are conceived of as distributed entities. Their elements are derived from the variable predicates of a process model and the objects for which those predicates are currently true. In the graphical notation the actual *marking* of a place represents the current extension of a variable predicate. The state given in the example intuitively means that there are currently two public modules named **n1** and **n2** whose contents are still undefined, and we have two authors **a1** and **a2** with **a1** being allowed to update module **n1** and **a2** being allowed to overwrite both **n1** and **n2**; further, there are three private module versions two of which, **p1** and **p2**, are intended to become new public versions named **n1**, whereas **p3** might replace public modules externally known by name **n2**.

Each development state together with the rules of change schematically defined by actions determines the set of possible future states. Transitions between development states are caused by occurrences of instances of actions that are concurrently applicable (see [10] for a formal definition of these concepts). One of the possible future states of our example is shown in Fig. 3. It was caused by occurrences of the changes `release-module(a1,m1,p1)` and `release-module(a2,m2,p3)`. These changes might have happened concurrently according to the given behavior specification. In contrast to this, two other changes that were possible at the initial state, `release-module(a1,m1,p1)` and `release-module(a2,m1,p2)`, mutually exclude each other as they "fight" for the same object named `m1`.

The set of possible states and state transitions is, as usual for Petri nets, defined by the *initial marking*.

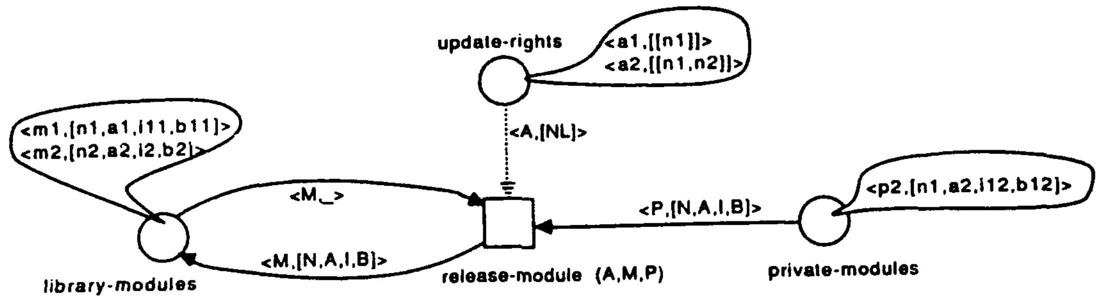


Figure 3: A possible future development state of the process model in Fig 2

4 Formalizing a Rapid Prototyping Process

In this section we develop a PNP model of a rapid prototyping process that supports evolutionary software development by interactive, computer-aided construction of executable prototypes from reusable software components. The model makes previous informal descriptions of this prototyping approach [12] more precise and concrete in that it provides suitable abstractions of software objects and captures causal dependencies and independencies among the actions of the process model. The PNP model also provides a better framework to develop an effective prototyping methodology, improve the functionality of the prototyping support environment [13], and control the proper use of its tools. Two different views of the process model are presented separately in Fig. 4 and 5 to simplify the understanding of the overall process, localize modification, and ease its further elaboration.

First we define some of the object and data types whose instances are involved in the rules of change specified in Fig. 4. These types refer to software concepts presented in [12]:

```
object req-def:  (sysname:name, description:text)
object operator:  (opname:name, spec:specification, state:eval-state)
record specification:  (inputs,outputs,statevars:[name-type])
vpredicate system-requirements(req-def)
           released-psdl-designs(operator)
action     construct(req-def,specification)
           modify(operator,specification)
           refine(operator,req-def,implementation)
           analyze(operator)
```

The process model view presented in Fig. 4 illustrates the principal idea of rapid prototyping to iteratively construct, modify, and refine a series of prototypes, each providing

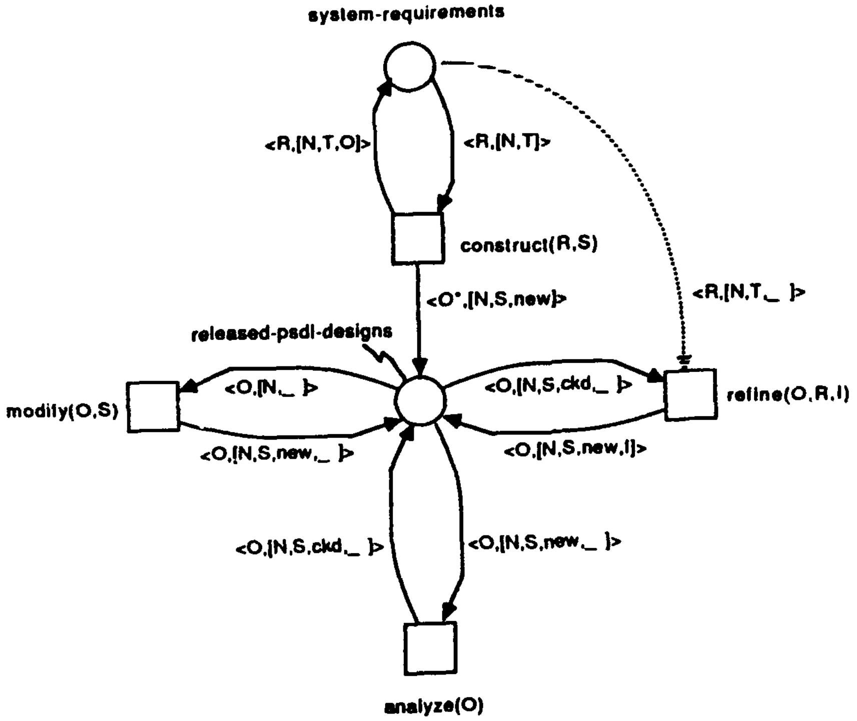


Figure 4: Constructing prototype designs from requirements specifications

the platform for validating and improving previous requirements definitions and design decisions.

At the given simplified abstraction level we do not want to formalize to what extend, for example, the text describing the requirements for a specific system component symbolically named N determines the specification of a newly constructed operator realizing these requirements. We just wanted to explicate certain relationships concerning names and references among objects. Looking more carefully at the net labelings, we recognize that certain variables denoting attribute values of objects after a change has occurred are not bound to attribute values existing before that change happened. An example is variable S which appears as argument of action modify and construct. It represents information which cannot be derived from the prehistory of the objects involved but has to be supplied by user of an action. Here the variable represents an arbitrary operator specification which redefines the spec attribute of the operator object changed by an instance of these actions. The *information flow* represented by such variables allows us to deal incomplete knowledge in such a way that at least its typical structure and its effect on the the behavior of a process model can be defined.

The process model in Fig. 5 shows another distinguishing aspect of the prototyping process model supporting the PSDL approach [12]. It reveals the role of prototypes to

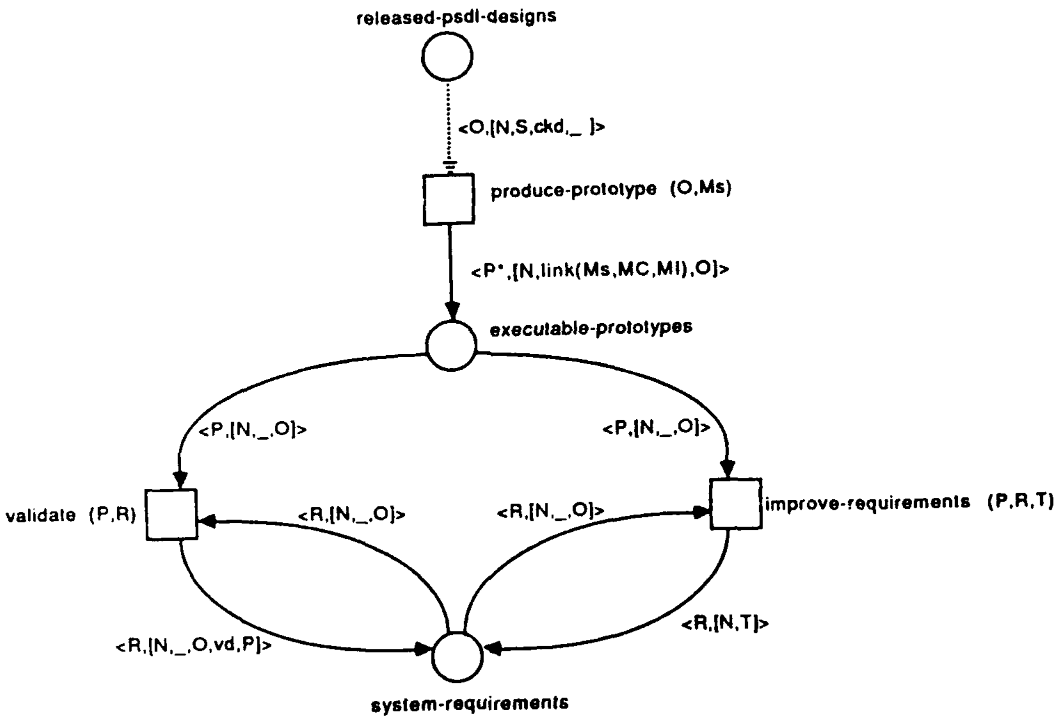


Figure 5: Constructing and evaluating executable prototypes

provide executable models of a proposed system which can be demonstrated to users and customers to validate and improve requirements specifications and design decisions prior producing production code.

5 Horizontal and Vertical Decomposition of PNP Models

One of the primary difficulties in modeling software processes is conceptual complexity. Conceptual complexity can be reduced when the dynamic behavior and the objects of a software process can be composed from independently constructed parts and can systematically be refined. Hierarchical process descriptions are supported by most of the new process models. But horizontal compositions in the sense of combining the parts of a modularized process model are still underdeveloped.

The PNP model approach provides constructions to consistently merge process models representing separate, partially overlapping views of a larger development process. These constructions allow the process designer to

1. synchronize the merged views and connect open information flow lines by identifying actions,

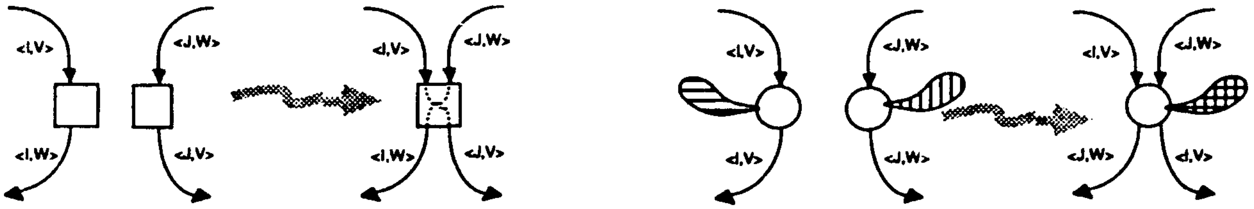
2. combine behavioral alternatives covered in separate views by identifying places and

forming the union of their initial marking, and

3. define new functions operating on objects from different views.

The context conditions to apply these construction and their formal semantics have been developed in the framework of a formal specification language for distributed and concurrent systems [9, 10] and can easily be adapted to PNP models.

Using these constructions the two process models shown in Fig. 4 and 5 can be combined by merging their common places labeled system-reqirements and released-psdl-designs. The implicit effect of the combination constructions on the behavior of the merged parts is graphically depicted below:



The PNP model also supports stepwise refinements based on substituting actions by subnets whose border only consists of actions, substituting places by subnets whose border contains only places, and abstract implementations of object and data types. Such refinement and implementation concepts have been studied in [14] for the related specification formalism with particular emphasis on defining correctness suitable criteria which provide the basis for verification tools. An example of an action refinement is given in Fig. 6. It shows that action produce-prototype can be implemented by two actions working concurrently on appropriate extracts of the object which is input to the abstract action.

6 Semantic Issues and Conclusion

Graphical representations of software concepts have certain advantages in conveying information to human readers but often lack a sufficiently precise semantics to be amenable to formal analysis, verification, and reasoning. One of the strengths of Petri nets is that they provide a simple graphical notation which is easy to comprehend even by non-experienced readers with a strong mathematical background.

The PNP model aims at exploits the comprehensible graphical notation of Petri nets and their precise causality-based execution semantics. It appears relatively easy and straightforward to provide a formal Petri net semantics of the PNP modeling approach by adaptin' the formal definition of the Petri net based specification formalism we have

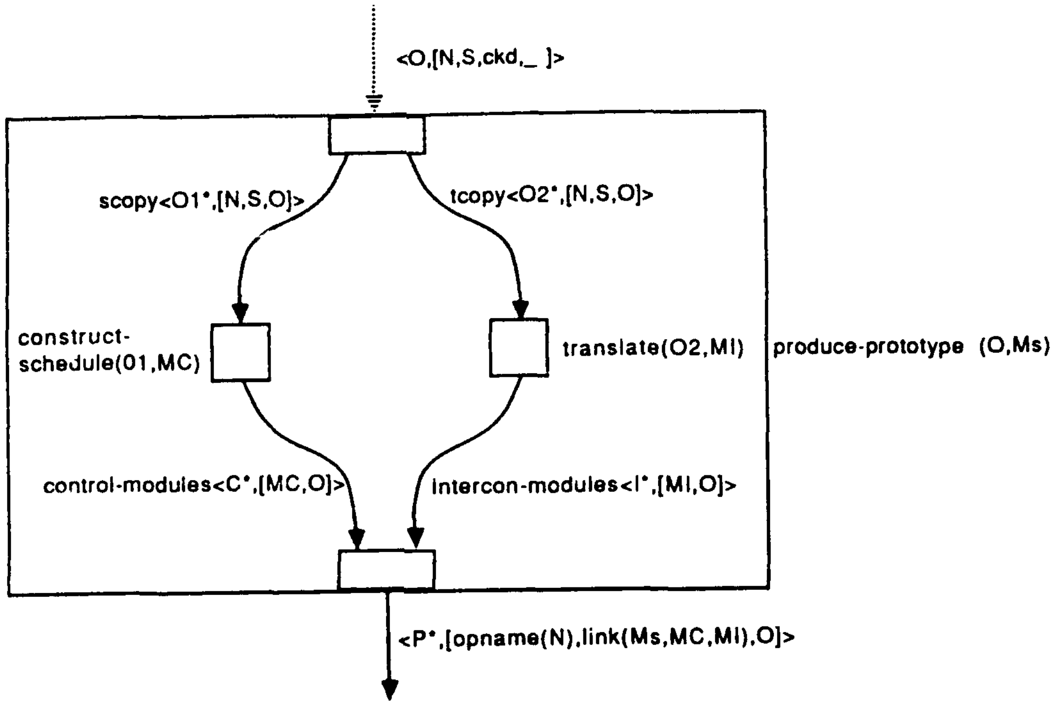

<O,[N,S,ckd,_]>
scopy<O1*,[N,S,O]>
tcopy<O2*,[N,S,O]>
construct-schedule(01,MC)
translate(O2,MI)
produce-prototype (O,Ms)
control-modules<C*,[MC,O]>
intercon-modules<I*,[MI,O]>
<P*,[opname(N),link(Ms,MC,MI),O]>

Figure 6: Refining an action of the process model in Fig. 5

developed earlier [10] to the new concepts introduced here to accommodate specific requirements of software process modeling.

The advantage of such a Petri net semantics would be that theorems, calculi, and validation methods for Petri nets can be reused to support consistency checking, liveness and safeness analysis, verification of the correctness of refinements [14], and invariant analysis techniques [5] for PNP models, too. Liveness and safeness analysis techniques, for example, would help to ensure the continuity of development activities and to prevent overload situations prior to executing a given process model. Or algorithms that generate and analyze the reachability structure of Petri nets might be adapted to support reasoning about behavioral possibilities and inherent facts of a process model.

The Petri net semantics also provides the basis for process model animation using a symbolic simulator for high-level Petri net specifications [2] which redistributes and rewrites objects according to the specified rules of change. Symbolic executions might help to get insight into the behavior of a the specified process and investigate the effects of alternative procedures prior to the actual execution.

References

[1] B.W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, pages 61–72, May 1988.

[2] M. Christ-Neumann, B. Krämer, H. H. Nieters, and H. W. Schmidt. The case environment graspin - user's guide. Technical Report ESPRIT Project GRASPIN 37/1,. GMD, 1989.

[3] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, 1988.

[4] A. Finkelstein. "not waving but drowning": Representation schemes for modelling software development. In *Proceedings of the 11th Annual International Conference on Software Engineering*, pages 402–404, Pittsburgh, Pennsylvania, May 1989.

[5] Hartmann Genrich and Kurt Lautenbach. S-invariance in Predicate-Transition nets. In Anastasia Pagnoni and Grzegorz Rozenberg, editors, *Applications and Theory of Petri Nets*, number 66 in Informatik-Fachberichte, pages 98–111, Berlin, Heidelberg, New York, Tokyo, 1983. Springer-Verlag.

[6] Hartmann J. Genrich. Net theory and application. In H.-J. Kugler, editor, *INFORMATION PROCESSING 86*, pages 823–831, Amsterdam, The Netherlands, 1986. IFIP, Elsevier Science Publishers B.V.

[7] Hartmann J. Genrich. Predicate/Transition nets. In W. Brauer, W. Reisig, and Rozenberg G., editors, *Petri Nets: Central Models and Their Properties*, number 254 in Lecture Notes in Computer Science, pages 207–247, Berlin, Heidelberg, New York, 1987. Springer-Verlag.

[8] W.S. Humphrey and M.I. Kellner. Software process modeling: Principles of entity process models. In *Proceedings of the 11th Annual International Conference on Software Engineering*, pages 331–342, Pittsburgh, Pennsylvania, May 1989.

[9] Bernd Krämer. *SEGRAS* - a formal language combining Petri nets and Abstract Data Types for specifying distributed systems. In *Proceedings of the 9th Annual International Conference on Software Engineering*, pages 116–125, Monterey, California, March 1987.

[10] Bernd Krämer. *Concepts, Syntax and Semantics of SEGRAS - A Specification Language for Distributed Systems.* GMD-Bericht. Oldenbourg Verlag, München, Wien, 1989.

[11] Bernd Krämer and Heinz-Wilhelm Schmidt. Object-oriented development of integrated programming environments with ASDL. *IEEE Software*, January 1989.

[12] Luqi. Software evolution via rapid prototyping. *IEEE Computer*, pages 13–25, May 1989.

[13] Luqi and Y. Lee. Interactive control of prototyping processes. In *Proc. COMPSAC 89*, Orlando, September 1989.

[14] Heinz-Wilhelm Schmidt. *Specification and Correct Implementation of Non-Sequential Systems Combining Abstract Data Types and Petri Nets.* GMD-Bericht. Oldenbourg Verlag, München, Wien, 1989.

DISTRIBUTION LIST

(1) Defense Technical Information Center 2
Cameron Station
Alexandria, Virginia 22304-6145

(2) Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Monterey, CA 93943

(3) Center for Naval Analysis 1
4401 Ford Avenue
Alexandria, VA 22302-0268

(4) Director of Research Administration 1
Attn: Prof. Howard
Code 012
Naval Postgraduate School
Monterey, CA 93943

(5) Chairman, Code 52 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

(6) Chief of Naval Research 1
800 N. Quincy Street
Arlington, Virginia 22217

(7) National Science Foundation 1
Division of Computer and Computation Research
Attn. Tom Keenan
Washington, D.C. 20550

(8) Naval Postgraduate School 100
Code 52Lq
Computer Science Department
Monterey, CA 93943

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1 AGENCY USE ONLY *(Leave blank)*

2 REPORT DATE
October 1990

3 REPORT TYPE AND DATES COVERED
Final

4 TITLE AND SUBTITLE
EVOLUTIONARY SOFTWARE ENGINEERING AT THE NAVAL POSTGRADUATE SCHOOL (NPS), MONTEREY

5 FUNDING NUMBERS
PE: 0602234N

6 AUTHOR(S)
Luqi, Assistant Professor of Computer Sciences

7 PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Computer Science Department
Monterey, CA 93943

8 PERFORMING ORGANIZATION REPORT NUMBER

9 SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Technology
Arlington, VA 22217

National Science Foundation
Washington, DC 20550

Office of Naval Research
Arlington, VA 22217

Naval Ocean Systems Center
San Diego, CA 92152-5000

10 SPONSORING/MONITORING AGENCY REPORT NUMBER
NOSC TD 1924

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document contains five papers that explore evolutionary software development. The five papers prepared by Computer Science Department of the Naval Postgraduate School are titled as follows:

1. Models for Evolutionary Software Development;
2. Multi-Level Software Analysis and Testing in Evolutionary Software Development;
3. Software Evolution via Prototyping;
4. Software Analysis and Testing Through Prototyping; AND
5. Petri Net-Based Models of Software Engineering Processes, (Keywords: real time, [illegible] maintenance, petri net models))

14. SUBJECT TERMS
software analysis
software evolution
software engineering

15. NUMBER OF PAGES
98

16. PRICE CODE

17 SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18 SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
SAME AS REPORT